LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Design and Implementation of a Packet Switched Routing Chip

MIT / LCS / TR-482

August 4, 1994

Christopher Frank Joerg

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

The author was supported in part by a graduate fellowship from the National Science Foundation.

This report was originally published as the author's Masters thesis.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Design and Implementation of a Packet Switched Routing Chip

Christopher Frank Joerg

MIT / LCS / TR-482
December 1990

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.
The author was supported in part by a graduate fellowship from the National Science Foundation.

This report was originally published as the author's Masters thesis.

Design and Implementation of a Packet Switched Routing Chip

Christopher Frank Joerg

Technical Report MIT / LCS / TR-482
December 1990

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Monsoon is a parallel processing dataflow computer that will require a high bandwidth interconnection network. A packet switched routing chip (PaRC) is described that will be used as the basis of this network. PaRC is a 4 by 4 routing switch which has been designed and fabricated as a CMOS gate array. PaRC will receive packets via one of its four input ports, store the packet in an on-chip buffer, and eventually transmit the packet via one of its four output ports. PaRC operates at 50 MHz, and each port has a bandwidth of 800 Mbits per second. Each input port operates asynchronously and has enough buffering to store four packets. The buffering and scheduling algorithms used in PaRC were designed to provide high utilization of the available bandwidth, while providing low latency for non-blocked packets. In addition, PaRC provides a mechanism whereby a processor can quickly receive an acknowledgment when a message it sent has been received. Although the design of PaRC has been driven by the needs of Monsoon, PaRC has been designed to be suitable for a wide variety of communication networks.

Key Words and Phrases: Interconnection Networks, Monsoon, Packet Switched Networks, Packet Buffering Algorithms, VLSI

Acknowledgments

I would like to thank all the members of the Computation Structures Group, especially Andy Boughton, my thesis supervisor, who has been a tremendous help both while I was working on PaRC and while I was writing this thesis. Despite having a million things to do, he was always willing to do what he could to help out. Thanks also to Ralph Tiberio, Jack Costanza, Ken Steele, Greg Papadopoulos, Brian Scott, Paul Bauer, and Constance Jeffery for their help and friendship.

And, of course, many thanks to my wonderful parents.

Contents

List of Figures

Chapter 1

Introduction

Multiprocessing is becoming prevalent as a way to increase the price-performance ratio of high performance computers. An important aspect of a multiprocessor is how its processors are interconnected. Some parallel machines simply connect their processors together by a shared bus. This is a very simple scheme that provides for quick message transfers and will often allow existing uniprocessors to be used with few modifications. But this method cannot provide enough bandwidth for more than a few processors; instead, interconnection networks are often used. These networks use switching elements to send messages between processors. Messages are sent from the originating processor, through one or more switching elements, and then to the destination processor. Since communication must now go through the switches, the latency of inter-processor communication has been increased. But the bandwidth is much greater since many messages can be sent simultaneously.

There are many types of switching elements and many types of networks that can be built out of those elements. The type of network used in a machine has a great impact on how the machine performs, and on what types of algorithms the machine can run well. PaRC is a routing switch designed to form the network for the Monsoon dataflow machine [10][9], but it has been kept general enough to be used in other types of networks as well.

Monsoon is a parallel processing dataflow computer currently being developed. A dataflow machine is one which executes using a data-driven model of computation.

Programs on a dataflow machine consist of dataflow graphs. Each node of the graph represents an operation to be performed; each arc in the graph signifies that the data produced by the node at the head of the arc is used by the node at the tail. The model of computation is called data-driven because an operation is free to be executed whenever it has all of its data. This is very different from a traditional von Neumann machine in which the operation to be executed next is selected based on which operation has just been completed. Since a data-driven model imposes fewer constraints on the ordering of operations, it is a promising model for parallel machines.

A dataflow processor operates by consuming and producing packets of data known as tokens. A token represents a piece of data flowing along an arc of a dataflow graph. A token is simply a piece of data connected to a tag. When a processor consumes a token it uses the tag portion of the token to determine what operation to perform on the data portion. After performing the operation, it uses the results to produce zero or more new tokens. In a single node dataflow machine, a processor will eventually consume all the tokens that it produces. In a multi-node dataflow machine such as Monsoon, one node may send a token to another, causing the second node to concurrently perform part of the computation. In this way, many nodes may cooperate on solving a single problem. The language Id [8] is a parallel language designed with dataflow processors in mind. From this language a compiler can directly generate dataflow graphs. These graphs allow the executable code to retain the parallelism that was inherent in the original algorithm. This allows a dataflow machine to exploit many opportunities for parallelism that conventional machines do not. To support this parallelism, a high bandwidth network is needed to interconnect the processors. The rest of this paper will describe the design and implementation of PaRC, a packet switched routing chip which will be used as the basis for this network.

1.1 Monsoon Requirements

Although PaRC was designed to be flexible enough to be used in many types of machines, its main goal was to provide a network for Monsoon. Since PaRC was optimized to work with Monsoon, we will first look at the requirements Monsoon places on any proposed network.

The first requirement is that the network provide high bandwidth to and from each node. Nodes will be either a dataflow processor or an I-Structure memory board [13]. It is noted in [9] and [1] that for compiled scientific code, 30-40% of the instructions executed by a processor will be either a store or a fetch. Most, if not all, of these will require sending the store or fetch through the network. [9] goes on to say that after including other causes for network traffic (*e.g.,* argument passing), on average half of the instructions executed will generate a token for the network. Since Monsoon is intended to be a high performance processor, a high bandwidth network is a necessity.

Also, the latency of the network should be kept to a minimum. Most algorithms go through periods where the potential parallelism is large, and periods where it is small. During periods of high parallelism, most processors will have plenty of tokens to process. So even if they have several pending memory requests, they will have other operations to execute while the requests are being processed. The opposite may be true during periods of low parallelism. A processor may send off a memory request and have very little to do until the request is answered. Since the time spent waiting for a reply may directly increase the execution time of the program, we want the network latency to be as short as possible. The latency of a network is a function of many factors, such as the configuration of the network, the latency of the network switch, and the amount of traffic in the network. Since during low parallelism traffic may be light, and since we are especially concerned with latency during periods of low parallelism, we want to pay particular attention to minimizing the latency during light traffic. In particular, we want to minimize the latency of non-blocked packets. By "latency of a non-blocked packet," we mean the delay from when the first word of a packet is received to when it is transmitted, assuming that the packet does not have to wait due to other packets using the output it needs.

Monsoon will use its network solely for passing around tokens. Given that tokens are of a fixed sized, the messages in the network will all be of the same size as well. Additionally, a Monsoon processor will sometimes need to know when a packet it sent has been received. The network should provide some means to acknowledge that a packet has arrived at its destination. In PaRC this is supported by allowing special packets called "Circuit Switched Packets." Also, the number of nodes in Monsoon may vary from just a few nodes to many hundreds. The network must be able to support all sized machines.

Lastly, let us emphasize that even though PaRC is being designed to support Monsoon, this is not its only goal. PaRC will be flexible enough to support many different types of networks.

1.2 Network Overview

Before designing PaRC, we first had to determine what type of network we wanted to build. Based on the above requirements, a number decisions about the network can be made.

The first is to make the network a packet switched network. A packet switched network (also called store-and-forward) is one in which messages are sent as a unit from point to point in the network. Switches have several inputs which are used to receive messages (packets), and several outputs which are used for sending packets. When a packet is sent to a switch, that switch stores the packet in a buffer. If the output that the packet needs to go to is available (*i.e.,* the output is not currently sending some other packet), the packet will be sent out through that output. If the output is unavailable, then the packet is blocked and, it will stay in the buffer until it can be sent out. A switch will accept packets as long as it has room to buffer them, regardless of whether or not the packets outputs are available.

An alternative to packet switching is circuit switching. In a typical circuit switched network, messages are transmitted by making a complete connection through the network from the sender to the receiver. The message is sent along this path, passing right through the switching nodes.[1] When making the connection, the header of a message moves from switch to switch, establishing a connection from the sender to the header's current location. If the header reaches its destination then there is a complete path from sender to receiver, so the message will be successfully sent. However, when the header enters a switch, the output it wants to use may already be in use. When this occurs the connection is not made, and the sender will have to make another attempt to send the message.

The main reason for choosing packet switching over circuit switching is the high throughput that is needed. A "link" is a connection which sends data to and from a switch. Depending on the machine, these links often make connections between boards or even between racks. Links are often the most expensive components of a network. Given a limited number of links, we need to make the best possible use of their bandwidth. A packet switched network will make better use of its bandwidth than would a similar circuit switched network. If a message can currently be sent only part way to its destination, a packet switched network will send the message as far as it can. A circuit switched network would not, thus wasting bandwidth that the packet switched network uses.

A disadvantage of packet switched networks is that they often have a longer latency than circuit switched networks. At each stage of a network, a packet may have to wait while other packets use the link it needs. Additionally, even if the packet does not have to wait for other packets, the switch may wait until the packet is fully received before it sends the packet to the next stage. This cause of latency can be lessened by allowing the switch to send out a packet before it has fully arrived. This is known as streaming. When this is done, and a packet's path is clear, the latency for a packet switched network will be comparable to that of a circuit switched network.

Another important characteristic of a network is its topology. The topology

[1] Circuit switching is often compared to making a phone call, whereas packet switching is compared to sending a letter through the mail.

of a network describes how the elements of a network are connected. An n-cube topology[11] (also called butterfly) was chosen for Monsoon. Figure 1.1 shows a network made of 2x2 butterfly switches.

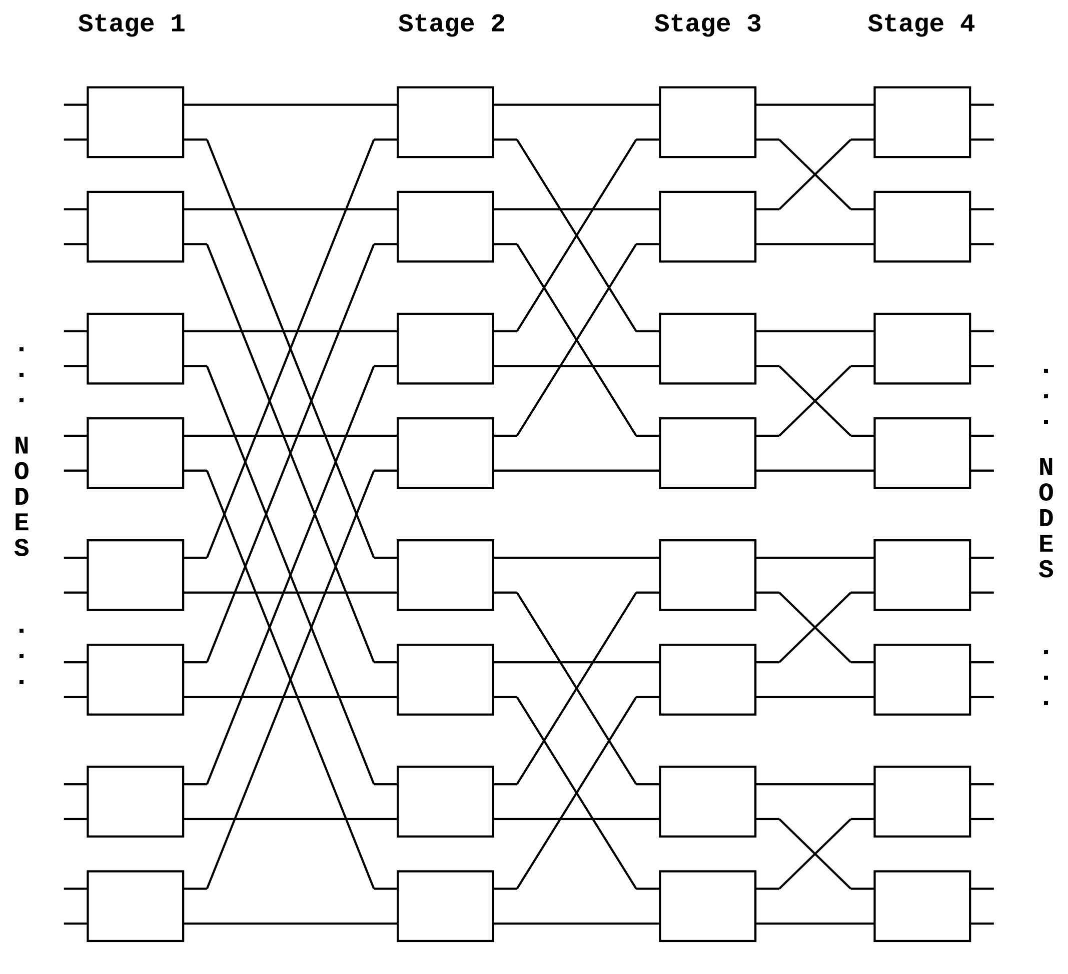


Figure 1.1: A Butterfly Network using 2x2 Switches.

An important feature of this topology is that any size network can be made out of the same switches. For networks such as the hypercube (which is used in the Connection Machine [15] and the Cosmic Cube [12]), the number of connections to a switching node is a function of the number of nodes in a network. Since the number of nodes in Monsoon will vary greatly, it is useful to have a switching node whose complexity will not increase as the size of the machine increases.

Another benefit of the butterfly topology is that it allows a message to go between any two nodes while passing through only a small number of switches. This is important since each node that a message passes through adds to its latency. In a butterfly network, a message will need to pass through only $O(\log N)$ switches, where N is the number of nodes being connected. This is also true of the hypercube, but is not true of networks such as a mesh. A mesh uses fixed sized switches, as does a butterfly, but a message may pass through $O(\sqrt{N})$ or $O(\sqrt[3]{N})$ switching nodes to get to its destination.

In the mesh and hypercube networks mentioned above, each switching element was associated with a node. These are called direct networks. The Monsoon network will be an indirect network. In this network only the switches in the first and last stages will be connected to nodes. All other switches will be connected only to other switches. An advantage of a direct network is that each node has several nodes which are very close to it (*i.e.,* significantly closer than the average node). This can be taken advantage of when writing programs. By trying to get nodes to communicate primarily with nodes they are close to, a programmer may be able to reduce the average distance messages need to travel, and thereby speed up the program. In Monsoon there is no sense of locality between processors. In other words, when one node needs to send a message, it is equally likely (or nearly so) that that message will go to any of the other processors.

A large Monsoon system will be spread out over many boards in many different cages. Providing a synchronous clock over such a large machine would be a very difficult task.[2] Since the nodes of this system are independent and may operate asynchronously, we did not want to provide a synchronous clock just for the network. For this reason each PaRC will operate asynchronously. This means that packets coming into PaRC from different places will all be transmitted from separate clock domains. The cost of synchronizing these incoming packets is that the latency of a packet is slightly increased. Since the networks will be spread out over many boards, some of the connections to and from PaRC may be quite long. This will affect the

[2] [16] shows some of the difficulties of providing a synchronous clock, and shows one way of overcoming them.

flow control mechanism. PaRC should work well on all sized links, including both short links between PaRCs on the same board, and long links going between racks. Lastly, the network should also preserve the ordering of packets. If two packets have the same sender and the same receiver, the packets should arrive in the same order in which they were sent.

1.3 Road Map

The work in this report is a continuation of the work begun in my bachelor's thesis[6]. The ideas in that thesis have been expanded upon and used in the design and fabrication of a PaRC chip. The rest of this report describes the design and implementation of PaRC. The next chapter begins with an overview of PaRC and then describes some of the key ideas that had the greatest impact on the design and performance of PaRC. Chapter 3 describes some of the implementation details of each section of PaRC. Chapter 4 describes the process of generating the test vectors which are used to test newly fabricated chips. The final chapter gives some conclusions and mentions some changes that future versions of PaRC could incorporate. The appendix contains the PaRC User's Guide. This guide contains the information users need to know when including PaRC in their system. Since this guide is meant to stand on its own, it duplicates some of the information contained in the body of this document.

Chapter 2

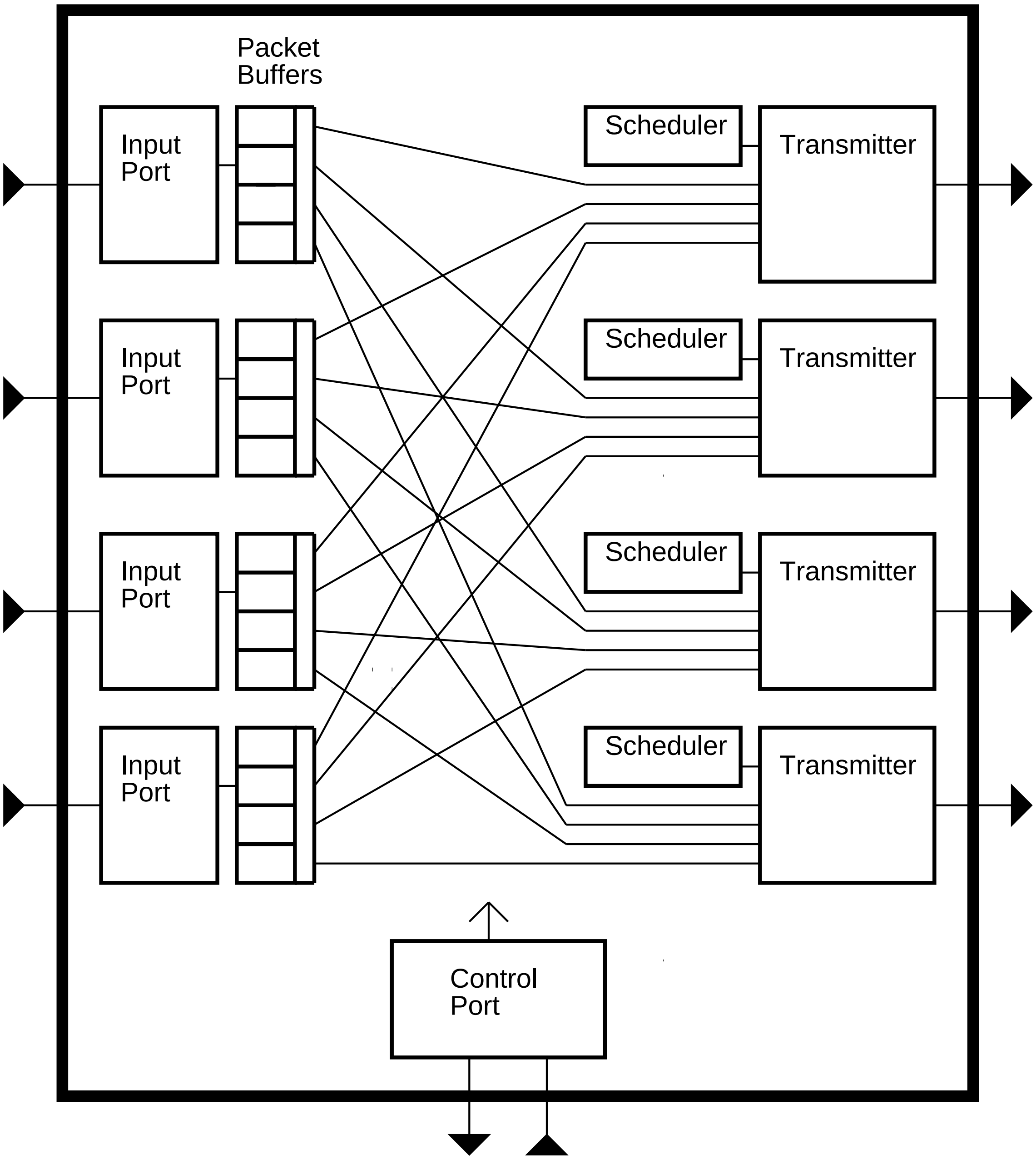
Design of PaRC

2.1 Overview of PaRC

PaRC is a 4 by 4 routing switch on a chip. It receives packets via one of its 4 input ports, writes them into an on-chip buffer, and eventually sends them out via one of its 4 output ports. This section will provide an overview of the major components of PaRC. Figure 2.1 shows these components in a top level diagram of PaRC. A more detailed description of these components can be found in Chapter 3.

The first major component is the input port. This component is responsible for receiving packets, checking them for errors, and writing the packets into memory. The input port must also determine on which output port a packet should be sent. It then makes a request to that output port's scheduler, telling the scheduler in which buffer the packet is being stored. The input port must also notice when its memory is filling up, and then notify the sender not to send any more packets. The memory of each input port is composed of 4 separate buffers, each of which can store exactly one packet.

Each output port is made up of two components: a scheduler and a transmitter. The scheduler keeps track of all the packets which need to use the output port and chooses which packet will be transmitted next. If two or more of the waiting packets were received via the same input port, the scheduler must guarantee that they are



The major datapaths and I/Os of PaRC are shown

Figure 2.1: Top Level View of PaRC

transmitted in the same order in which they arrived. The transmitter is the component which sends packets off the chip. When the transmitter is ready to send a packet, it first finds out from the scheduler where the next packet is being stored. The transmitter then reads the packet out of its buffer and transmits it to the next stage of the network. If the packet is a circuit switched packet (*i.e.,* a packet which requires an acknowledgment), the transmitter must also ensure that an acknowledgment is produced at the appropriate time. Circuit switched packets will be discussed in more detail in Section 2.4

The control section has two main functions. One function is to keep statistics on the performance of the network. The other is to provide an interface by which the performance of PaRC can be controlled and monitored. Through this interface all of PaRC's programmable features can be controlled (such as how routing is done). This interface can connect to a network control system which will allow the operation of the network to be controlled by software.

2.2 Buffer Utilization

A design decision that greatly affects the performance of a packet switched network is how much buffering each switch has and how that buffering is used. Typically, store and forward networks store messages coming in over the same input in one long fifo (first-in-first-out) buffer associated with that input. PaRC does not do this. Instead, it splits the memory associated with each input port into four separate buffers, each of which is large enough to hold exactly one packet. The principle reason for doing this is to maximize the chip's throughput.

If the memory was organized as one large fifo queue, then it would only be possible to read out the packet at the top of the queue. When that packet was blocked, nothing could be read out of that fifo, even if the other packets in the fifo were not blocked. This puts a severe limitation on the throughput of the network. Assuming that the top of each port's fifo has a packet which is heading for a random destination, then on average only 2.7 of those 4 packets would be unblocked. This means that even if the links entering a chip can supply a packet whenever room is available, the utilization rate for links leaving that chip would be only 68%. This means that one-third of the each link's bandwidth would be wasted!

This rate can be improved by looking at more than just the top packet. If the first packet is blocked we could look at the next packet and send that packet if it is not blocked. If we do this then the best case utilization rate rises from 67% to 80%. (The best case utilization rate is defined to be the average utilization of output ports if all buffers always have a packet. This is not a measure of how well we expect to do, but is a limit on how well we could possibly do.) If we can look at all four packets and send the first one that is not blocked the best case utilization rises to 90%.

The ability to read from any buffer can be used to our advantage even more. We can give each buffer its own output circuitry, thus allowing each buffer to be read independently. It then becomes possible to read out two (or more) buffers from the same input port at the same time. This permits each output port to read from any buffer that has a packet for it, regardless of which other buffers are currently being read. By doing this, the best case link utilization rate for PaRC rises to 99%. Another advantage of adding output circuitry to each buffer is that it makes it faster to begin to read out a packet. Since the buffer always knows which word will be read next, it can begin the read of that word on the cycle before it is needed. This helps to minimize the latency of packets.

Two properties of PaRC are exploited in reaching this best case utilization rate. First, we make use of the fact that all of the packet memory can be put on-chip. If packets were stored externally it would be very expensive to use this scheme because it would greatly increase the number of chips needed for packet buffering. It would require a larger number (at least one per buffer) of small memories (or fifo memories), as well as additional external circuitry to multiplex together the results. (Using multiported memories could reduce this cost somewhat.)

Also, if the packets were not of fixed size, but their sizes varied greatly, this scheme would be very inefficient. To use this scheme, we would have to make each buffer large enough to hold the largest allowable packet. For smaller packets, part of the buffer space would be wasted. In a single fifo scheme, memory space is not wasted because each packet takes up only as much room as it needs. So given a fixed amount of memory, the separate fifo scheme would buffer significantly fewer packets. Although if most packets were close to the maximum packet size, this loss would be relatively insignificant.

These are significant improvements. Not only do they increase the throughput greatly, but they also decrease latency; packets which would have been stuck behind other packets can now be sent as soon as their output port is available. These improvements are successful because they help to prevent output ports from being idle while there are packets waiting to use them.

These improvements, which were first described in my bachelor's thesis, are related to the recently published Virtual Channel Flow Control scheme of Dally [3]. This scheme is mostly concerned with networks where packets may be large enough so that switches will not buffer entire packets. The concept of virtual channels are used to achieve results similar to PaRC by restricting a packet to use only a portion of the buffering in an input port. This is done by dividing up the buffer space into several smaller separate fifos, and assigning each fifo to a different "virtual channel." When a packet is sent to a switch, the sender must associate the packet with a virtual channel that is not currently in use, and the packet can only use the buffer space associated with its virtual channel. If a packet becomes blocked it will fill up its buffer space, and the sender will stop sending the packet until more space is available for it. Since a packet can only use the space associated with its virtual channel, it does not fill up the rest of the buffer space in that input. This allows another packet to be sent to that input and stored into the buffers associated with a different virtual channel. This packet may take a different path than the earlier packet so it may not get blocked. This improves both the average throughput and latency of the switch. PaRC's buffering scheme is similar to a virtual channel scheme in which there are four virtual channels, and each virtual channel has enough buffering to store exactly one packet. PaRC's scheme is simpler since it does not deal with buffering only part of a packet. PaRC's scheme also requires less bandwidth overhead since the receiver does not have to be told what virtual channel the data is being sent on, and since separate flow control information is not needed for each virtual channel.

PaRC's buffering strategy greatly improves the chip's throughput and latency, but these improvements do come at a price. In addition to adding to the size of the packet buffers, this scheme makes the scheduling problem much more complex. The scheduler must still ensure that packets following the same path do not get out of order. More precisely, packets that arrive through the same port and which will go out the same port must be sent out in the same order in which they were received. This is easily done when there is a single fifo buffer, since packets can only be read out in exactly the same order they arrived. This is not true in PaRC; there is no way to tell which of two packets arrived first simply by looking at packet memory. The scheduling strategy will have to solve this problem.

2.3 Scheduling Strategy

The scheduling strategy had to be designed to take advantage of the large number of independent fifo buffers that each transmitter could read from. It also had to deal with the ordering problem outlined above. There are two types of schedulers that PaRC could have used: centralized and distributed. In the distributed method, each output port has its own scheduler; while in a centralized method there is one scheduler that makes the scheduling decisions for all the ports. A centralized scheduler has the advantage of more flexibility; it can easily deal with packets that can be sent to more than one output port. However, since it is more complex, it is slower and probably cannot schedule more than one or two packets each cycle. A distributed method was chosen so that ports would not have to sit idle for several cycles while waiting for the scheduler to get a chance to schedule them.

In a simple distributed scheme, each packet buffer has a request line to each scheduler and holds it high whenever it has a packet going to that output port. This strategy would not work in PaRC because it would fail to guarantee that packets traveling between the same ports are kept in order. Adding a timestamp to each request would not work because there is no limit on the amount of time that a packet could have to wait in a buffer.

The first idea on how to deal with this problem was to prioritize each of the packets in an input port. The packet which has been waiting the longest would have the highest priority. This worked as follows: When a packet arrived and all buffers were empty, it would be given the highest priority, 3. If another packet arrived before this packet left, it would be given a slightly lower priority: 2. (In general the priority given to an incoming packet is $(i-1)$, where i is the number of available buffers). Each time a packet is removed, all remaining lower priority packets would have their priority increased by 1. Note that if several packets were removed on the same cycle each of the remaining packets would have to be increased by an appropriate amount. On each cycle the input port would have to look at its four buffers and choose the highest priority packet going to each output port. The schedulers would then choose between the packets chosen by the input ports. This scheme would work but the logic for it would be very complex and, more importantly, slow; thus adding extra cycles to the latency of non-blocked packets.

To simplify (and thus speed up) scheduling, a way to use the advantages of single fifo buffering is needed. This is done by putting a fifo queue in each scheduler; but instead of buffering packets, these fifos only need to buffer pointers to packets. When a new request arrives, a pointer to the requesting packet is added to the bottom of this scheduling fifo (s-fifo). Choosing the next packet to be transmitted is as simple as reading the top of the s-fifo. This scheme makes it possible to start reading out a packet on the cycle immediately after its request has been made.

There is one inefficiency we must tolerate with this scheme. To guarantee that none of these s-fifos fill up, each scheduler's s-fifo must be large enough to store a request from every input fifo. So there is four times as much s-fifo space as could ever be in use at one time. This is bearable because each s-fifo location is small (5 bits: 4 to point to the packet, and one to indicate whether or not the packet is a circuit switched packet). The total size of all the s-fifos is equivalent to approximately two packets worth of buffering.

PaRC's buffering and scheduling schemes, both of which were described in my bachelor's thesis, are related to the later work of Frazier and Tamir. Their dynamically-allocated-multi-queue (DAMQ) buffer [4] is a way to configure and control an input port's memory in order to reduce output port contention. In a DAMQ buffer the memory is split into fixed sized blocks. Packets are stored using one or more full blocks. Several logical queues of these blocks are maintained in the DAMQ buffer, and the DAMQ is able to move blocks between queues. One of these queues is a list of free blocks; the rest are queues of blocks which contain packets headed for the same output port. At a given time the head block of any one queue may be read out of memory. As with PaRC this increases the performance of a machine by allowing packets to leave an input port in a different order than they arrived. However the DAMQ buffer is not as efficient as PaRC since only one packet can be read out of an input port's memory at a time.

By maintaining queues in each input port, a switch using DAMQ buffers solves the problem of keeping packets in fifo order. In a DAMQ buffer, each queue has its own head and tail pointer; each block of memory also has its own pointer which is used to establish its position in a queue. The state of these pointers describes the current state of the queues. The DAMQ moves blocks between queues by manipulating these pointers. However moving a block between queues (*e.g.,* from the free list to an output port's queue) is a complex task that takes three cycles. (It actually takes 6 cycles since the control of these pointers is shared between logic for incoming and outgoing data.) PaRC does a similar job by using a scheduler fifo. Since PaRC uses an actual fifo its control is much simpler. In each cycle an item can be added to and removed from the queue. The price for this speed and simplicity is that extra memory space is needed since at most one-fourth of s-fifo memory is used at any one time. Also, by placing the queue in the output port rather than the input port, PaRC provides first-come-first-served scheduling for packets which arrive via different input ports.

2.4 Circuit Switched Packets

2.4.1 Why They Are Needed

Circuit switch packets were added to PaRC so that a processor can send a packet and be sure that it has been received before continuing. Although most of the time processors do not care when their messages are received, there are a few cases where it does matter. One example is when memory is to be deallocated. An object in memory can be deallocated only if we know it will never be used again. In particular, when it can be proven that an object is only used within a certain block(s) of code, we would like to be able to deallocate that object after running that code.

To safely deallocate an object we must be sure that all reads and writes of that object have been completed. Since code will only complete after all the reads have returned, we know that once the code has run there are no outstanding reads. However, this is not true with writes. It is possible for a location to be written but never read; so just because the code has completed does not mean that all the writes have completed.

If all the writes, as well as the deallocation, were issued by the same processor, this would not be a problem. The deallocation would be issued after the writes and the network would guarantee that they arrived in the same order. But this is not always the case. Node A may send the writes to node B and then tell node C that the code block has completed. Node C may then send the deallocation to B. It is possible that it may take longer for a write to go from A to B than for the deallocation information to go from A to C to B, this would cause the write to arrive after the object has been deallocated.

To prevent this from happening node A must not tell C that it is done until all of its writes have completed. One way for A to know that the write is complete is to do a read of the location. But this is slow, as it would require at least the full network transit time of two packets. This would also increase network traffic, since it requires sending two extra packets. Another way is to do a write which requires the memory to send back a packet to acknowledge the write. However, this would still be slow and would still increase network traffic (although only by one extra packet). Instead we use packets which can generate an acknowledgment without sending back an additional packet. These are called circuit switched packets. In the above example, processor A can send the write as a circuit switched packet, so it will get an acknowledgment when the packet arrives. Once A knows that its write has arrived at B, it can inform C that it has completed. If C then sends a deallocation message to B, that message could only arrive after the write.

2.4.2 How They Work

The original idea for circuit switched packets was that as they traveled through the network they would hold on to the links they crossed, thus creating a connection from the sender to the receiver. This connection would be similar to those in circuit switched networks, hence the name "circuit switched" packets. When the packet reached its destination, an acknowledgment signal would be sent back along this connection, thereby informing the sender that its packet had arrived. This signal would also break each connection after crossing it. This scheme had the drawback that links could be blocked for long periods of time during circuit switched packets. During most of this time the link would be idle, since once the packet was sent, nothing else could be sent until the connection was broken.

This scheme was greatly improved by allowing normal packets to be sent across the link while waiting for the acknowledgment signal. All that must be done is to keep track of where the packet came from. So now a circuit switched packet is sent the same way as any other packet, the only difference is that when the packet leaves a PaRC chip, the transmitter keeps a "back pointer" to the port which the packet came from. These back pointers lead from the packet's current location back to the sender of the packet. When the packet reaches its destination an acknowledgment signal is generated which uses these pointers to find its way back to the sender and inform it that its packet has arrived.[1] This does not eliminate all blockage due to circuit switched packets. If a circuit switched packet needs to be sent out through a port that is already waiting for an acknowledgment, the port will be blocked until the acknowledgment for the first circuit switched packet arrives. If the second circuit switched packet were sent, then when an acknowledgment signal arrived, there would be no way of knowing which packet the acknowledgment was for.

Another way to reduce the disruption of circuit switched packets is by minimizing the time spent waiting for the acknowledgment signal. This is done in two ways. The first is minimizing the time it takes for an acknowledgment to be routed through a PaRC chip. We do not wait for the acknowledgment to be latched in and synchronized before we pass it on. Instead as soon as a PaRC chip receives an acknowledgment, it begins to send it on.

The second way to minimize the time spent waiting for acknowledgments is by generating each acknowledgment as soon as possible. One way to acknowledge circuit switched packets is to have the receiver generate an acknowledgment once it receives the packet, but it is not really necessary to wait this long. It could be done sooner by taking advantage of the property that once a packet is being sent to the receiver, no new packet could possibly get there before it. This means we can have the PaRC chips in the last stage of the network generate an acknowledgment as soon as they begin sending a circuit switched packet to the processor/memory node.

The acknowledgment can be generated even sooner if we take advantage of the fact that each node receives packets from only one PaRC chip. Once a packet has been received and its request placed on the appropriate s-fifo, there is no way that any new packet can get ahead of it on that s-fifo. Since that is the only way to get to the receiving node, no packet not already on the s-fifo can get to the receiver before this packet. This means we can generate the acknowledgment as soon as a packet's

[1] Each link has a dedicated wire that is used for transmitting this acknowledgment signal. This wire transmits data in the opposite direction as the rest of the link.

request is stored. Taking this one step further, we can have the next to last PaRC chip generate the acknowledgment as soon as it sends out the circuit switched packet. This can be done since once the packet has started to be sent out, it is guaranteed to be put on the next s-fifo within a short, fixed period of time. When this is done, the final stage of PaRC chips should treat all packets as normal packets.

2.5 PaRC Interface

When sending packets, the output ports of PaRC will have to send those packets according to a specified protocol. Of course, the input ports will have to receive packets according to this same protocol. This protocol will define the interface between PaRC chips. This interface is important because it will influence the number of wires needed in a link (and hence its cost) and the amount of usable bandwidth we get out of those wires.

Each link will consist of 16 bits of data, accompanied by a clock. Each link needs its own clock because, as mentioned earlier, each part of the network operates asynchronously. It is assumed that the rising edge of the clock occurs while the data is stable. Since PaRC can operate at 50MHz, this gives us a raw bandwidth of 800 Mbits per second per port.

The data that the processor needs to send are messages which are 144 bits long. Since our datapath is 16 bits wide, this data takes up 9 words of our packet. In addition, a packet also contains 2 extra words which are used to check for errors.[2] Finally, we need information on how to route the packet. This information will be placed in the first word, the header, since we want to be able to start sending out a packet soon after it arrives. This gives us a 12 word packet whose format is shown in the following table.

[2]Optionally, one or both of these words can be used for data instead of error checking.

WORD	USE
0	Header word
1	Data word 0
2	Data word 1
3	Data word 2
4	Data word 3
5	Data word 4

WORD	USE
6	Data word 5
7	Data word 6
8	Data word 7
9	Data word 8
10	CRC word 0
11	CRC word 1

PaRC Packet Format

The above information describes what packets look like, but it doesn't say how an input port can determine when a packet begins. One possible way is to have an extra wire which provides a frame bit. This bit would go high only during the first word of a packet. This would work but would be a very inefficient use of the wire. Instead PaRC uses one of the bits in the header (the uppermost bit) as a Start-Of-Packet (SOP) bit. When an input port is not receiving a packet, it looks at the value of this bit. When the SOP bit is 0 then no packet is being started. When it becomes a 1 then a packet is being started, and this word is its header. The next 11 words must then be the rest of the packet. During these words the bit position used as a SOP bit is used as a normal data bit. In this way only one bit per packet is "wasted" as overhead. If we had used a separate wire for a frame bit, we would have wasted 12 bits of bandwidth per packet.

Once the 12 words of a packet have been received, the input port begins looking at the SOP bit again to see when the next packet begins. There is no need for an idle word between packets; an output port may begin to send a new packet immediately after the previous one has completed. This also helps to make maximum use of our bandwidth.

In addition to the SOP bit, the header of a packet also contains up to 15 bits of information which will be used to determine how the packet will be routed. The use of these bits will be described in the section on routing (Section 2.8). Optionally, one of these bits can be used to indicate whether or not this packet is a circuit switched packet. The format of the header is:

Header Format		
Bit 15	Bit 14	Bits [13..0]
1	CSP/ROUTE-DATA.14	ROUTE-DATA.[13..0]

When an output port is not transmitting a packet, it will transmit one of 2 specified idle patterns. (Of course both of these patterns have a 0 in the SOP bit.) The output port will either alternate between these patterns (to keep the output data bits changing) or can always use the same pattern. (This will cut down on the power usage.) Specified idle patterns are used to allow an input port to detect virtually all link transmission errors. How this is done will be described in the following section.

2.6 Error Detection

The PaRC interface is designed to detect virtually all link transmission errors. If an error were to go undetected, Monsoon could produce an incorrect result. We want to minimize the possibility of this happening. It is also important that we detect where the error occurred, so that we can take steps to prevent it from reoccurring. If we only did end-to-end checks (*i.e.,* checking the validity of packets as they enter and leave the network) we would not be able to do this.

2.6.1 Types of Errors

There are three types of errors that an input port can detect on a link. These are room-error, CRC-error, and idle-error.

Each input port only has enough buffers to store 4 packets. A room error occurs when a packet arrives and there is no room for it.[3] When this occurs the input port will receive the entire packet and simply discard it.

Error checking is done on packets by use of a Cyclic Redundancy Code (CRC). As each data word of a packet is received, it is accumulated into a checksum. PaRC uses

[3]The flow control mechanism should prevent this from occurring. See section Section 2.7.

a 32 bit checksum and accumulates values using the CRC32 polynomial. The header and the 9 data words are accumulated to produce a checksum; this checksum is the 32 bits that should appear in the final two words of the packet. A CRC error occurs when the checksum computed by the input port does not match the CRC at the end of the packet. Using this code there is less than a 1 in 10^9 chance of an incorrect packet being mistaken for a correct one. In addition there are certain types of errors which will always be detected. More details on the CRC are given in Section 3.3.

As mentioned earlier, PaRC's output ports always output one of two specified patterns, called idle patterns, when they are not sending a packet. When an input port is not receiving a packet it checks to see that each word it receives is one of the idle patterns. An idle error occurs when a word is not part of a packet and is not one of the idle patterns. By checking for CRC and idle errors, the input port is able to detect virtually all link transmission errors.

Here are the possible transmission errors, and how they will be detected:

- *Error occurs inside of packet (*i.e., *any bit except SOP bit).* This will cause the checksum to be incorrect, thereby creating a CRC error.

- *Error turns SOP bit from 1 to 0.* The input port will expect the word to be an idle pattern, since a packet is not starting. Since it probably will not be, an idle error will occur. In addition, the remaining words in the packet will either cause idle errors, or begin a packet which will have a bad CRC.

- *Error occurs in SOP bit of idle pattern (*i.e., *turns SOP bit from 0 to 1.)* This will create a packet which will have a bad CRC, causing a CRC error.

- *Error occurs in other bits of idle pattern.* This will give an idle error (but will not cause any loss of data).

This makes it extremely unlikely that an undetected error will occur. And when errors do occur, it will be easy to pinpoint their location since we can tell where the error first appeared.

Since the network will be spread out over many boards, the links between PaRC chips will need to go from board to board. To reduce the number of wires that must be sent between boards, and to reduce the chance of errors on these connections, the Data Link Chip [2] was designed. This chip will take the 16 signals from from an output port, multiplex them into 4 values, and transmit them differentially at 4 times the speed of PaRC. A DLC will also be at the receiving end of the link to receive these signals. It will demultiplex these values and send them to an input port, along with a clock. The transmitting DLC can also detect errors and report them to the PaRC chip.

When the network is initialized, each transmitting DLC will synchronize itself to the output clock being produced by the port it is connected to. This will let it know when it is safe to sample the data coming from the output port. This output clock is generated from PaRC's main clock (ICLK). Since ICLK is generated by the DLC from the DLC's clock, the DLC should stay synchronized to the data out clock. If the data-out clock were to drift from its original position in relation to the DLC clock, it is possible this synchronization could be lost. The main reason that the clock might drift is the changing speed of the PaRC circuitry as it heats up. Synchronization should not be lost even when PaRC moves across its entire allowable temperature range. Nevertheless, the DLC continually checks to ensure that synchronization has not been lost.

When a DLC detects a possible problem with its synchronization, it can report one of two errors: LINK-ERR and LINK-GRAY. The difference between these errors is that when a link-gray occurs no data has been lost. When a link-err occurs it is likely that bad data has been transmitted. (Note that if bad data was transmitted, it should be detected by the input port that is receiving data from that link.) When one of these errors is detected the link should be taken out of use, resynchronized, and put back in use. All of this can be done via software.

2.6.2 How Errors are Dealt With

When an error is detected we would like to

- Prevent any data from being lost, if possible; else minimize the amount of data lost.
- Determine where the error occurred.

To prevent data from being lost, or minimize the amount that is lost, when we detect an error we may want to stop using the connection on which the error occurred. To help do this, when PaRC detects an error, it will immediately bring its ERROR output high. The network control system can see this and stop the ports which are using that connection. This prevents any more data from being lost. Unfortunately it may take many PaRC cycles before this can be done.

Because of this PaRC can be programmed to go into "passive mode" when it detects an error. When in passive mode, PaRC will not begin to transmit any more packets. In addition, when in passive mode each of the input ports will tell the port which sends packets to it to stop sending packets. This should stop packets from being sent to or from the PaRC chip, thus limiting the amount of data that could be lost.[4] In particular, if the only errors recorded were from the link, then no data at all will have been lost. Software can tell PaRC to resynchronize the links and have PaRC exit from passive mode; the machine will then continue normally. Even if errors were detected by the input port it is possible that no data has been lost if the errors were caused by corrupted bits in idle patterns.

To help determine where the error occurred, PaRC keeps track of what errors it has detected. For the errors detected by input ports, PaRC keeps track of how many (0, 1, 2, or more than 2) of each error type was detected by each port. For each type of link error, PaRC keeps track of whether or not that error has occurred on each of the output ports. All of these values can be read via PaRC's control port.

[4]Although if the flow control line is giving erroneous values, packets may continue to be sent to the chip.

2.7 Flow Control

Flow control is needed between each transmitter and the input port that it connects to so that an input port does not receive more packets than it has room to buffer. The mechanism chosen for this is important because a sub-optimal mechanism could needlessly delay packets, which would both add latency, and reduce the throughput of the links. This problem is not trivial due to the delay in sending flow control information from the receiver to the sender. We may have to turn off the sender before the receiver is full so that the data sent after it is told to stop does not overshoot the amount of room available to store data.

The scheme used in PaRC has a simple basis. The input port produces an asynchronous signal called WAIT, which is sent to the output port. When this signal is low the output port can send packets. When it is high the output port should stop sending packets. (Of course, any packet it has already started to send should be completed.) This mechanism has the advantage of needing only one signal sent between transmitter and receiver. Also the logic to implement it is fairly simple. To produce WAIT the input port looks at all four of its packet buffers, and asserts WAIT only if none of them are currently ready to accept a new packet.

There is one major problem with this strategy. If it takes too long for the transmitter to receive the WAIT signal, it will begin sending another packet even though there is no more room. The length of the delay between PaRC chips is critical because this delay counts twice in determining if the WAIT signal will be received in time: A transmitter begins reading out a packet and then sends it out across a link. An input port receives the first word and if there is no room for any packets after this one it will assert WAIT. This signal is then sent back across the link to the transmitter and synchronized to its clock. This synchronized signal must go high soon enough to prevent the transmitter from sending a new packet.

Given the particulars of PaRC's implementation the link transit time of the data plus the link transit time of the WAIT signal must be less than 8 cycles (= 160ns) for this to work correctly. Considering the delay of the DLC and the wiring we expect to use, this will allow links over 30 feet long. Although this should be long enough for the links we expect to use, we still want PaRC to be able to support longer links.

An important feature of this scheme is that its performance was optimal; no other strategy could do better.[5] There are other schemes which could be used, which also give good performance, and will work on longer links. One possibility is to have the receiver send back a value giving the number of packet buffers currently available. However, this scheme would be very costly as it requires sending back several lines for data, and possibly a clock as well. A variation of this is to send back a pulse each time a packet buffer is freed. This allows the transmitter to determine exactly how many buffers the receiver has available. Since the sender would then know exactly how many buffers the receiver has available, this would also give optimal use of the packet buffers. This requires only a single signal but it has several problems, such as what happens if a transmitter misses one of the pulses. Also the logic becomes complex since, for example, several buffers can become available at once, but the input port can only send a pulse every other cycle.

To allow longer links, the input port can be programmed to generate a different wait signal called LWAIT (for Long WAIT). LWAIT is similar to WAIT except that it is asserted whenever there are only 0 or 1 available buffers. As soon as three buffers are full, the input port says to stop sending. If the link is long enough the transmitter will send out one more packet before it is stopped by the LWAIT signal. This is fine since LWAIT was raised while there was still one buffer available. Using LWAIT the allowable round trip transit time is increased by almost 12 cycles. This allows links much longer than will probably ever be needed.

The problem with using LWAIT is that its performance is not optimal since it sometimes wastes one of the buffers. This may occur in two places:

- When the number of available buffers drops from 2 to 1. LWAIT may go high in time to stop the next packet, thus leaving one of the buffers unused.

[5] It is optimal given the constraints of the fixed delay between sender and receiver and given that we do not know in advance when buffers will become available.

- When the number of available buffers rises from 0 to 1. There is now an available buffer, but no packet will be sent to it until a second buffer becomes available.

The first concern is not as much of a problem as it may at first seem. In light traffic, using LWAIT will not hurt much because the input buffers will rarely fill up enough to force LWAIT high. In heavy traffic, when a third buffer is put in use, LWAIT will be asserted. But since traffic is heavy other packets will have been waiting to use the link and by the time the rising LWAIT signal gets to the transmitter, the transmitter will have already begun to send another packet. So the fourth buffer will be filled anyway.

This will be true only on longer links, since on short links the LWAIT signal will be received before the transmitter begins to send the packet. By using either WAIT or LWAIT based on the length of the link we can minimize this problem. On links where the rising WAIT signal is guaranteed to be received in time we can use the normal WAIT signal. On links where the rising WAIT signal is guaranteed not to stop the next packet, LWAIT will be used and if there is a packet waiting it will be sent into the fourth buffer. It is only on the remaining links, those where the rising WAIT signal may or may not arrive in time, that we lose. (The uncertainty is due to the synchronization of the WAIT signal, and to variations in component delays.) Since the WAIT signal may not arrive in time we must generate the LWAIT signal. But if the signal does arrive soon enough, it may prevent a waiting packet from being sent into the fourth buffer.

To minimize the performance degradations of the above two concerns, PaRC can generate a modified LWAIT signal which differs from the normal LWAIT in two ways. The first modification is to delay the rising edge of LWAIT by two cycles. On links where the LWAIT signal may or may not arrive in time, this should increase the delay enough so that all four buffers can be filled. De-assertions of LWAIT are not delayed because when LWAIT goes low we want new packets to be sent as soon as possible. This has the side effect of reducing the allowable round trip transit time by two cycles. But even with this change we can still use links much longer than we will need.

The second concern above is that when all four buffers are full and one of the buffers becomes free, LWAIT stays high. It will not be de-asserted again until there are two available buffer spaces. A buffer is now being wasted since there is an empty buffer and nothing is allowed to be sent to it. There is a way to improve upon this as well. When the number of free buffers increases from 0 to 1, the transmitter is allowed to send one (and only one) more packet. This is done by de-asserting LWAIT for two cycles. When the transmitter notices that LWAIT is low it will begin to send a packet if it has any. It can only send one because LWAIT will be high again by the time the transmitter finishes sending the packet. If the transmitter does not have any waiting packets during the time when LWAIT is low then no packets will be sent, even if one arrives shortly thereafter.

For both of these modifications there is only a small window of time when a packet will be able to be sent into an otherwise wasted buffer. If no packet needs to be sent during that time, then the buffer will remain empty. So it is during moderate and heavy traffic that these modifications will be most effective; and that is exactly when they are most needed. In fact, if there are always waiting packets, then these modifications cause the LWAIT flow control strategy to give us optimal performance. It is only when the modifications provide a window in which a packet could be sent, and one is not, that this strategy is sub-optimal.

2.8 Routing

When a packet enters PaRC, the input port must decide which of the 4 output ports the packet should be sent out of. This is where the 14 (or 15) ROUTE-DATA bits from the header are used. Each PaRC chip can be made to look at two different bits of ROUTE-DATA. This allows networks of size $2^{14} = 16K$ to be supported; 32K if circuit switched packets are not being used.

The standard way that routing is done is to tell the PaRC chip which two bits of the routing data to look at. Packets going through that chip will then go to the port indicated by the two selected bits. With this routing mechanism, butterfly routing can easily be done by using the destination address as the routing data. Each stage of the network can then route based on consecutive bits of the address. Figure 2.2 shows one way this could be done in a butterfly network made up of 2x2 switches. Since the destination address is 011, the first stage of the network sends the packet out port 0, the second stage port 1, and the third stage also port 1. This is true regardless of where the packet enters the network. Routing can be done the same way with 4x4 switches except that each stage uses two bits for routing instead of one.

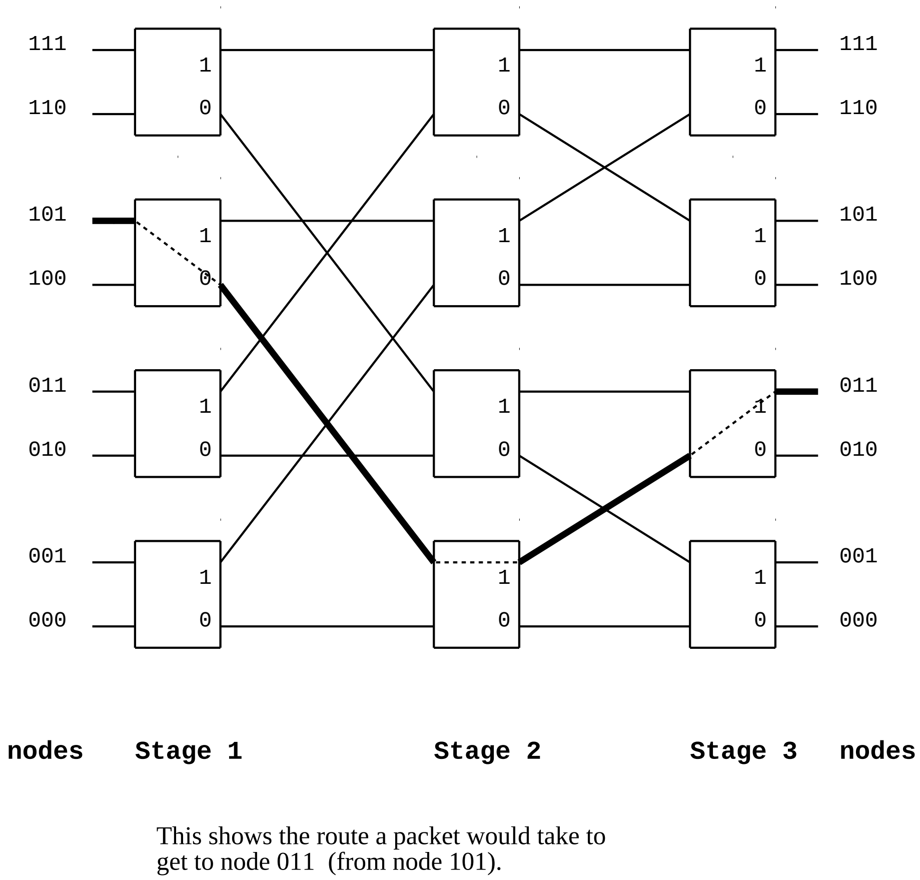


Figure 2.2: Routing in a Butterfly Network

PaRC is also able to do routing based on local congestion. In this mode only one bit of the routing data is looked at. This bit is used to determine if the packet should got to a top port (ports 3 and 2) or to a bottom port (ports 1 and 0). If it is determined that the packet should go to a top (bottom) port, PaRC will choose the top (bottom) port with the shortest queue of waiting packets. This is called Up/Down routing. Of course, this will be useful only in networks in which there are several paths that a packet can take to its destination, and in systems where a packet may go to any of a group of destinations.

PaRC can also act as two parallel 2x2 switches. Packets which enter through one of the upper two input ports are transmitted out through one of the upper two output ports. Which of the two output ports is used is determined by looking at one bit of the packet's routing data. Similarly, packets entering through one of the two lower ports are routed to a lower port. A variation of this routes based on congestion. Packets entering through an upper (lower) port are sent to the upper (lower) port with the shortest output queue.

There are some minor modifications to the network Monsoon will use that would be useful for some machines. Normally the only way one node can talk to another is to send a message that travels the length of the entire network. In many machines it is beneficial for each processor to have a portion of memory which it is closer to. Objects which will be frequently accessed by a processor will be placed in the memory close to that processor. Other objects can be distributed as before. Part B of Figure 2.3 shows one way this can be done. In this configuration, a processor and memory unit are bundled together with a PaRC chip. Each has its own port to and from the network, just as in the standard configuration. But since the PaRC chip has been added, the processor and memory can now communicate through a short path. This will reduce the latency of messages sent between the processor and the memory, and reduce the volume of traffic that is sent into the main network. By reducing the number of messages sent into the main network, we allow the network to support higher performance nodes.

In some machines it may be desirable to closely bundle three components. For example, the Epsilon-2 dataflow machine [5], being designed at Sandia National Labs, may bundle together a processor, a memory, and an IO unit. A way to produce such

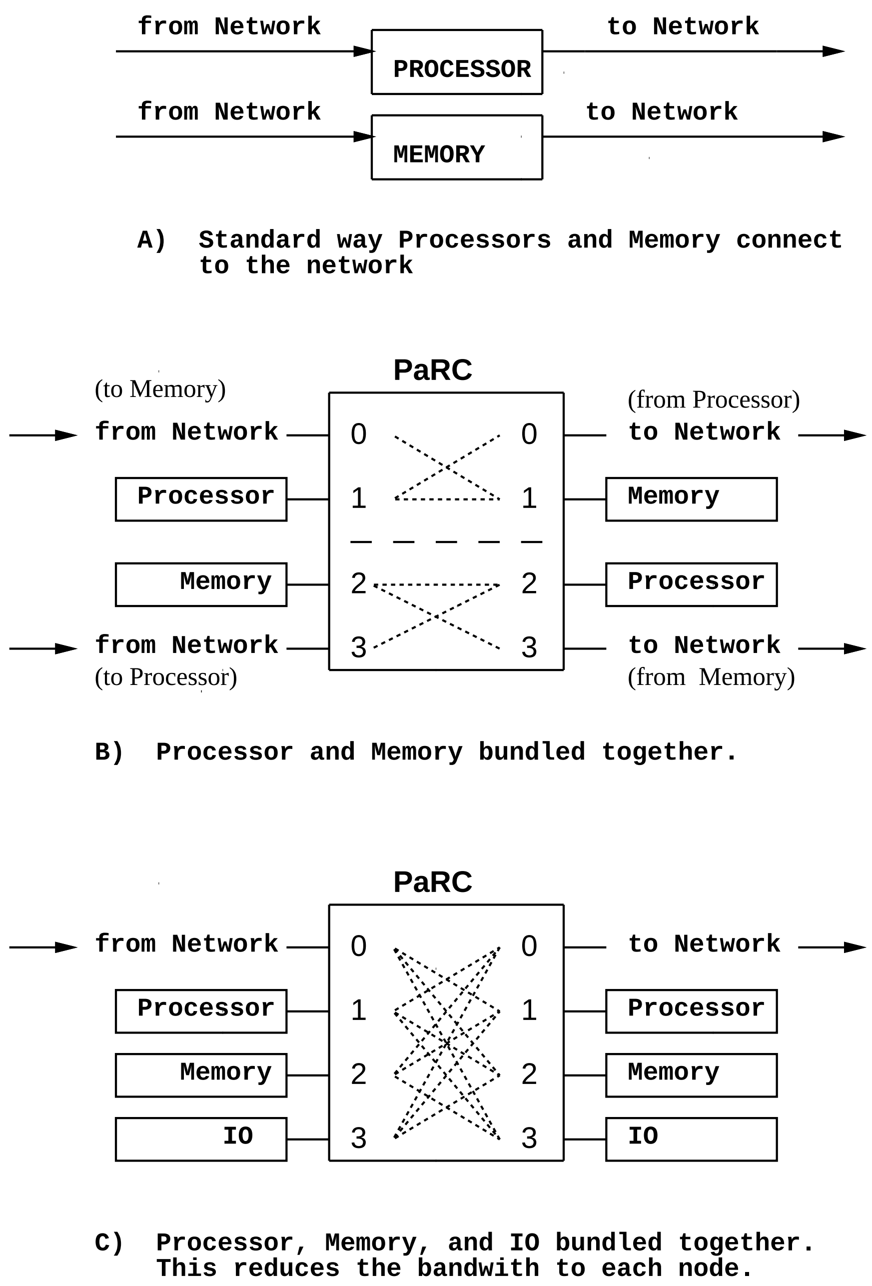


Figure 2.3: Several Ways to hook nodes into a network.

a configuration is shown in part C of Figure 2.3. This allows the three components to quickly communicate with one another. Since there is only one port to the network, it must be shared by all three components. This decreases the bandwidth that each has to the rest of the machine. So to use such a configuration it is necessary that a large portion of the communication is to one of the local nodes.

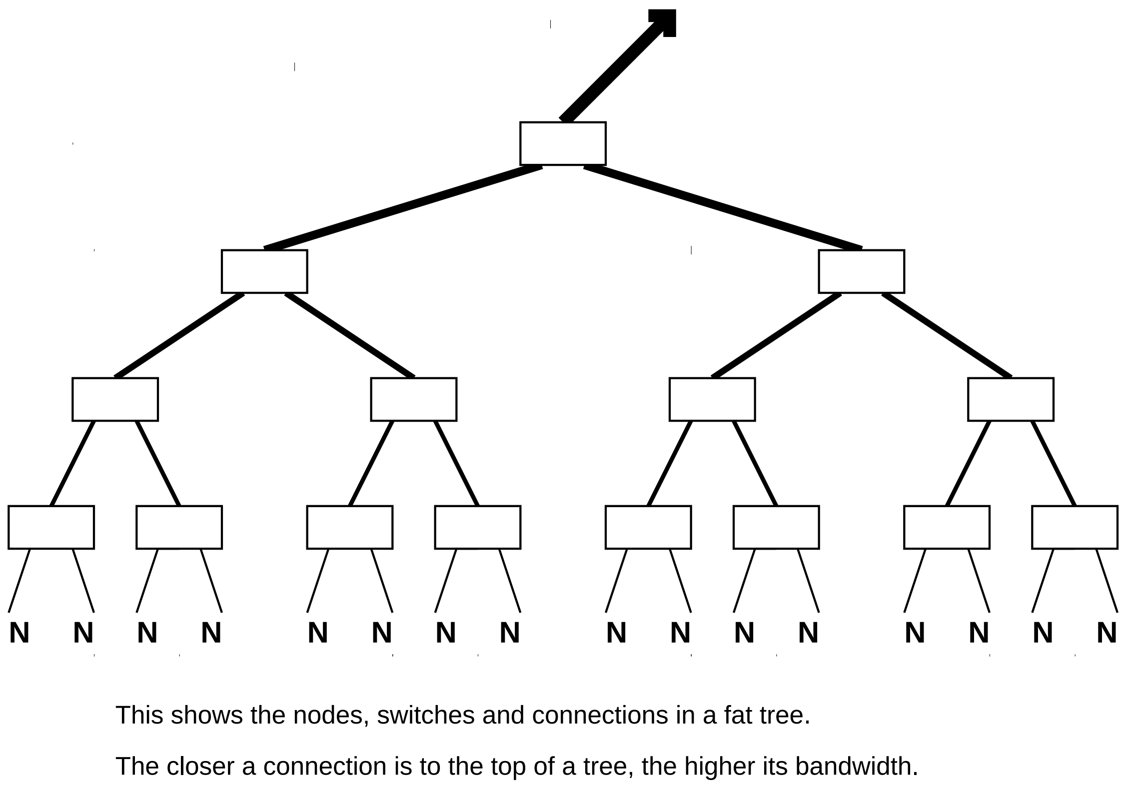


Figure 2.4: A typical fat tree

PaRC would be useful in many networks besides butterfly based networks. One example is the fat tree network [7]. A fat tree is a network with a topology of a binary tree. The leaves of the tree are the nodes (processor, memory ...); the nodes of the tree are the switches. Messages start at the leaves and move up the tree as far as they need to[6], then they go back down the tree. Figure 2.4 shows a portion of a fat tree network. Since more messages need to pass through the higher nodes of the tree, the

[6]Each time you move up a tree you double the number of leaf nodes that you are above. Once a message is above its destination it need go no higher.

bandwidth of each link often increases as you go up the tree. Often there are separate links for going up and down the tree, in this case you can think of the up and down portions of the network as being separate: A message first moves up the tree via the "up" network. Once it has gone far enough it switches to the "down" network and is switched down to the appropriate leaf. Figure 2.4 could represent either half of this network since they are symmetric. Not shown are the paths between the up and down networks.[7]

At each node in this network a one bit routing decision is made. On the trip up messages come in from two locations and the switch decides if the message should continue going up, or if it should be sent to the down tree. On the trip down, the switch decides which of the two sub-trees the message should be sent to. Since each stage routes based on one bit, we can easily make this network out of PaRCs. This is shown in Figure 2.5. In order to increase the bandwidth we will increase the number of PaRCs in a switching node. To double the bandwidth in a link we use double the number of PaRC chips at the switching node at the top of that link. (We could also increase by other amounts as well, but that is slightly more complex.)

When we double the bandwidth, each PaRC in a switching node will have one input from each subtree, two outputs to the next higher level and two to the down tree. (Switch the words *inputs* and *outputs* if this is a down tree instead of an up tree.) Packets may cross either of the two connections to get to the next level. On the up tree, PaRC's Up/Down routing mechanism would be used to send packets to the output with the shorter queue.

When we do not increase the bandwidth, each PaRC in a stage has one connection coming from each sub-tree. In addition, it has one output going to the parent and one going to the down tree (assuming we are looking at the up tree). Since only two inputs and outputs are used, only half a PaRC chip is needed. Since one PaRC chip can act as two parallel 2x2 routers, we could combine two switches. In Figure 2.5 this

[7]The top of the tree is not shown and would be handled differently. No switching node would be needed as any message coming up one side of the tree would always want to go down the opposite side of the tree.

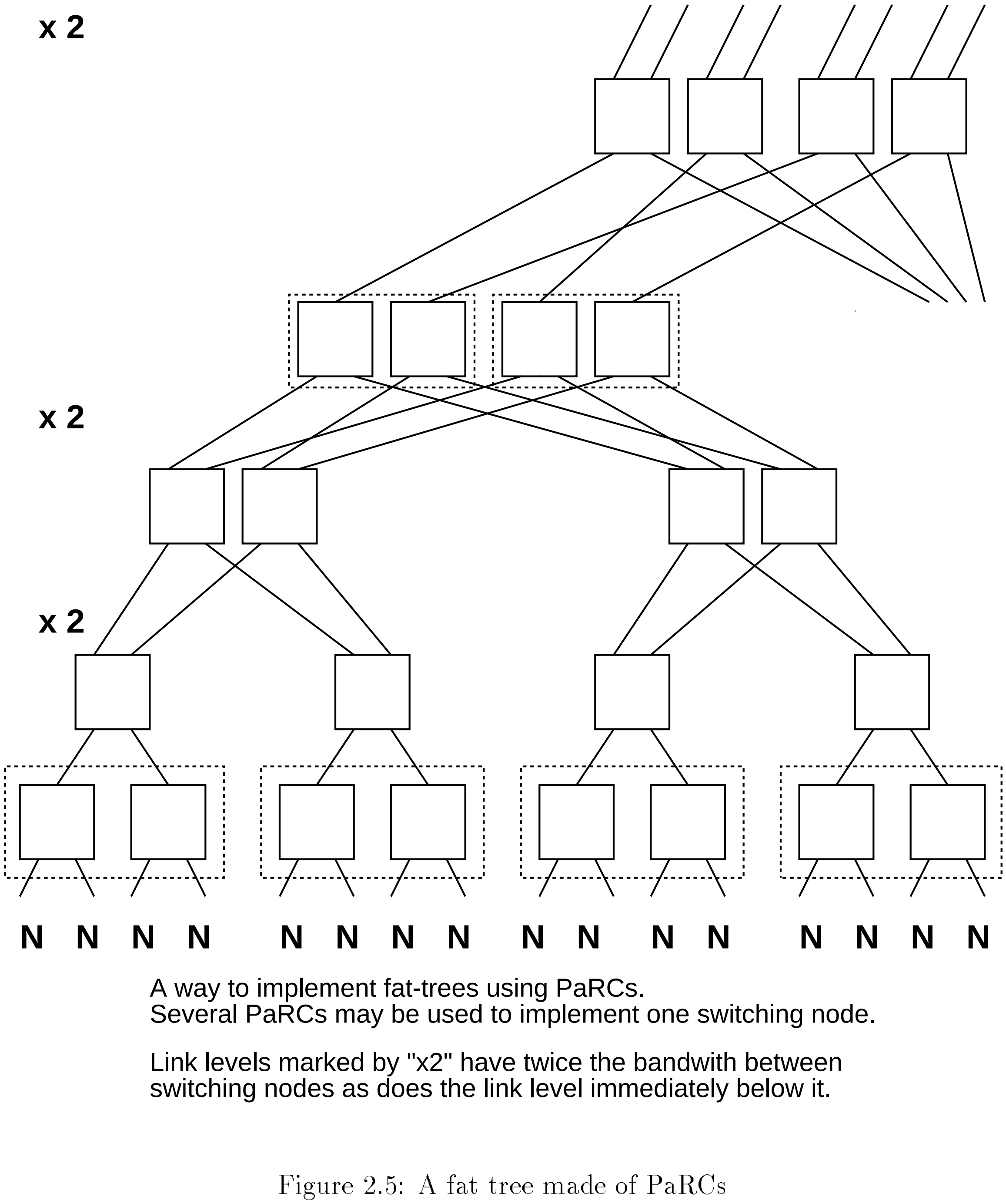


Figure 2.5: A fat tree made of PaRCs

is shown by dotted lines around two switches that could be combined into one PaRC chip. When we combine switches we may be able to make use of PaRC's Up/Down routing mechanism. This is because the outputs of the two switches are often going to the same places.

2.9 Control Port

PaRC needs a way by which its operation can be controlled. Rather than having this done by special control packets, PaRC has a dedicated control port. This port does not receive or send packets; instead it operates via a simple read-write protocol. This port is used to control various aspects of the operation of PaRC. It also allows information about the performance of PaRC to be read out.

The parameters of PaRC which can be controlled by this port include:

- How routing is done.
- Whether the CRC should be checked or generated, and how many words long the CRC is.
- What errors should be checked for.
- What is done when errors are detected.
- Whether or not circuit switched packets are allowed.
- Which ports are operating and which are turned off.

The performance values that can be read out include:

- What errors have occurred
- What input buffers are in use
- How long each s-fifo queue is

In addition this port keeps statistics on how the output ports are being used. These statistics will tell what percentage of the time each output port spends:

- Transmitting packets

- Idling with no packets to send
- Blocked due to WAIT being high
- Blocked due to Circuit Switched Packets

By looking at these statistics we can get a good idea of how effectively each output port is being utilized.

Chapter 3

Implementation of PaRC

3.1 The Technology

PaRC is designed in LSI Logic's LCA10000 CMOS compacted gate array series. The die used contains almost 75,000 gates. (A gate is defined as four transistors, the equivalent of one NAND gate.) Since routing is done over gates, typically around 40% of the gates on a chip are usable; the exact figure varies greatly depending on the design. PaRC uses over 33,000 gates.

On chip propagation delays for this technology are fairly fast. A two input NAND gate (ND2) in this series has a nominal low-to-high propagation delay of 0.6 ns. However, this is not as fast as it seems. This delay is increased due to the wire connecting this output to other inputs, and due to the loading of those other inputs. When the ND2 gate is driving 4 gates, its nominal delay may increase to about 1.5 ns. The above values are the nominal times. The actual delays can vary greatly based on process factors when the chip was fabricated, and the environmental conditions (*i.e.,* temperature and voltage) the chip is used in. The best case delays are about 1/2 the nominal delays, and the worst case delay are nearly twice the nominal. So the worst case delay for the ND2 mentioned above would be about 2.8 ns.

Of course other gates have different delays. The delay of a 2 input NOR (NR2) gate under the same conditions may have a worst case delay of 5 ns, fully one quarter of PaRC's cycle time! There are also high drive gates which are more effective at driving large loads. If the above NR2 gate were a high drive gate (NR2P), its worse case delay would improve to about 3 ns. But there is a catch: high drive cells are larger, and often have higher input loadings than the normal drive gates.

Since the delays depend so much on the wiring, exact delays are not known while the design is being done. Once the schematics are complete, a top level floorplan of the chip is done. This floorplan tells where on the die the top level blocks of the design should be placed. LSI Logic uses this as a guide when doing the complete layout of the chip. After the layout is done the exact wirelengths are known. These wirelengths are used to provide a much better estimate of gate delays.

The rest of this chapter will describe the implementation of the components which make up PaRC.

3.2 Packet Buffers

Each input port has its own FMEM block which is made up of four packet buffers, along with some multiplexing logic. The packet buffers each store exactly one packet. They can only be written by that one input port, but can be read by any of the output ports. The multiplexing logic is used to route the data to the appropriate output port. The four FMEM blocks account for about half of the gates used on the chip. A top level diagram of a single packet buffer is shown in Figure 3.1.

A major constraint that affected the design of the packet buffers was that all events had to be created due to the rising edge of the clock. The incoming clock may have been generated in the previous stage's PaRC chip, so it may be very unbalanced by the time it arrives on this PaRC chip. Since there is a very large variance in where the falling edge could be with respect to the rising edge, we could not use the falling edge of the clock to help generate any signals. Also since the speed of circuits varies greatly with process and temperature, we could not reliably use a delay element to give us a pulse that lasted only a portion of a cycle. This meant that all pulses had to last for one or more full cycles.

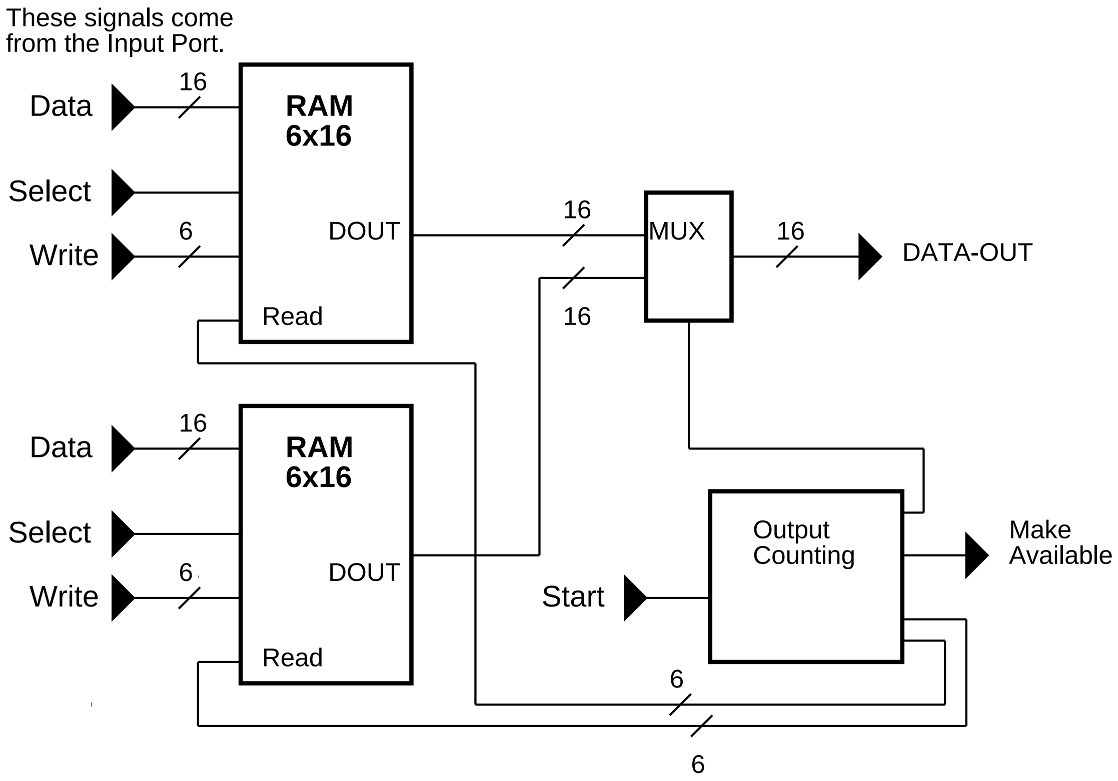


Figure 3.1: A Packet Buffer

In particular, this meant that the write pulse for the memory within the packet buffers had to be an entire cycle long. Because we needed to write the ram during consecutive cycles, and because the address and data would have to change based on the same edge as the write pulse, the memory had to be divided into several banks.

For these banks there was the choice of using pre-defined memory blocks, or designing the memory ourselves out of single bit memory cells. The prebuilt memory cells are specially laid out to optimize the use of chip area. So they are denser than a similar memory built out of discrete cells. Of course, since these are predefined they must be used as is. They do not allow the same flexibility as a specially designed memory would.

Using the prebuilt cells would have required a three cycle write phase with the write pulse occurring during the second cycle. In order to guarantee the setup and hold times for the address, the address would have to appear during the cycle before the write, and continue into the cycle after the write. Since signals can only change at the start of the cycle, the address would need to appear at the start of the first, and the write pulse could appear at the start of the second. The write pulse would not end until after the start of the third, so the address would have to remain valid throughout the third cycle.

By designing the memory ourselves, we can get more flexibility. Each packet buffer in PaRC is broken into two blocks, each with six 16-bit locations. Instead of a block having an address and a single write line, each location in a block has its own separate write line, and each block has a select line. A word is written whenever the word's write line and its block select line are both active. Since we do not have an address that must be stable both before and after the write pulse, this arrangement allows a bank to be written every other cycle. (Two cycles are needed because the data must be stable both before and after the end of the write pulse.)

Each packet buffer contains the logic necessary for reading packets out of the buffer. This allows a packet to be given to an output port as soon as a port needs it. Each word in a block has its own separate read line; a word will drive the block's output bus whenever its read line is asserted. The control logic guarantees that exactly one of each block's read lines is asserted at all times. The output of the two memory blocks are multiplexed together to provide the data output of the packet buffer.

The packet buffer has a START signal which tells it to start reading out a packet. The buffer waits until it receives the START signal, sends out the 12 words it has stored, and then waits until it receives another START signal. When not in the process of reading out a packet, a buffer has its two memory blocks reading out their first word. The output from the first of these blocks is selected by the mux. This allows an output port to read the first word of the packet during the same cycle in which it asserts START. On the next cycle the packet buffer flips the select input to the mux so that the second word is sent out. At the same time the read address for the first block is incremented, so that the third word can be sent out on the next cycle. In this way the value to be sent next will always be available before we need to send it.

When a packet buffer reads out a packet, it must also tell the input port that the buffer has been freed. Since the input port may be waiting for another buffer, the buffer should be released as early as possible. This can be done before the packet is completely read out because it will take several cycles before the input port can synchronize the buffer-available signal and start to write a new packet into this buffer. It is even safe to have another packet being written into the buffer while the current packet is being read out, but only if we can guarantee that no output port will try to read the new packet until after the first packet has been completely sent out. The signal to release the buffer is given when the eighth word of the packet is being read; the reasons why this is safe will be described in the next section.

The FMEM block, shown in Figure 3.2, contains the four packet buffers associated with an input port. The multiplexers in the FMEM block are used to help each output port get the data from the appropriate packet buffer. Each output port can read from any of the sixteen packet buffers on the chip. This means we need to multiplex the data from the 16 buffers so that an output port can read the data from the correct buffer. The FMEM block performs the first stage of this multiplexing. The data from each packet buffer goes into four 4-to-1 multiplexers. Each output port receives data from one of these multiplexers. The output port has direct control of the select lines for its multiplexer, so it can select the data from any of the four buffers. Putting the multiplexers in the FMEM block cuts down greatly on the amount of area needed for placing wires between the packet buffers and the output ports. Instead of having a 64 (= 16 x 4) bit bus going to all four output ports, there are four 16 bit buses each going to just one output port.

3.3 Input Port

The Input Port is broken up into a number of sub-blocks. The PACSTART block looks at the incoming data to determine when a packet starts. The PACCNT block counts

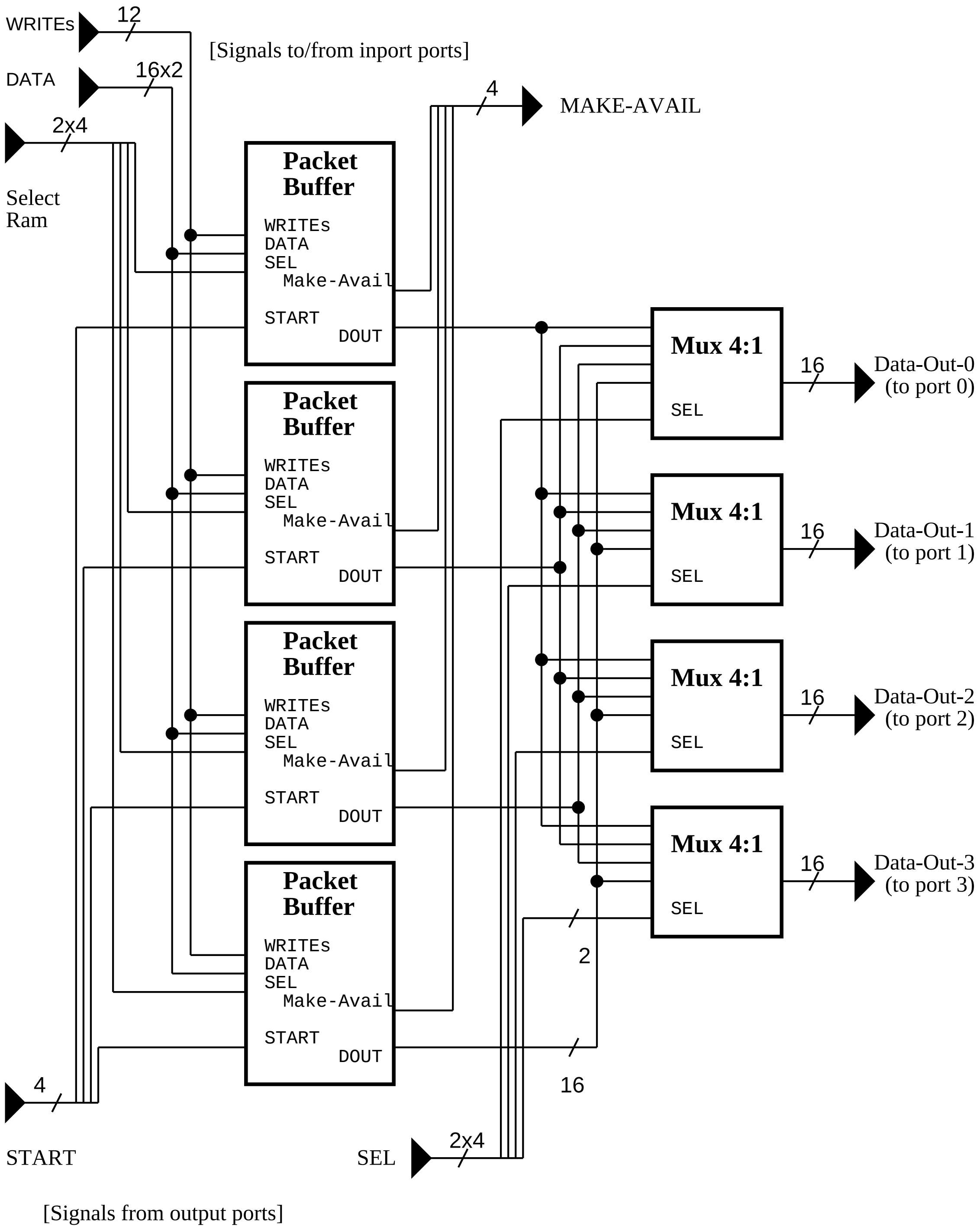


Figure 3.2: The FMEM Component

the incoming words of the packet and produces the write signals for the packet buffers, while the RDATA block supplies the data for the packet buffers. The AVAILB block keeps track of which buffers are available, and the WAITUNIT uses this information to decide when to assert Wait. The SELPORT block determines which port the packet should be sent to, and the MAKEREQ block makes a request to that port. Lastly, the CRC and IDLE blocks check for CRC and IDLE errors. Figure 3.3 shows a block level diagram of the input port.

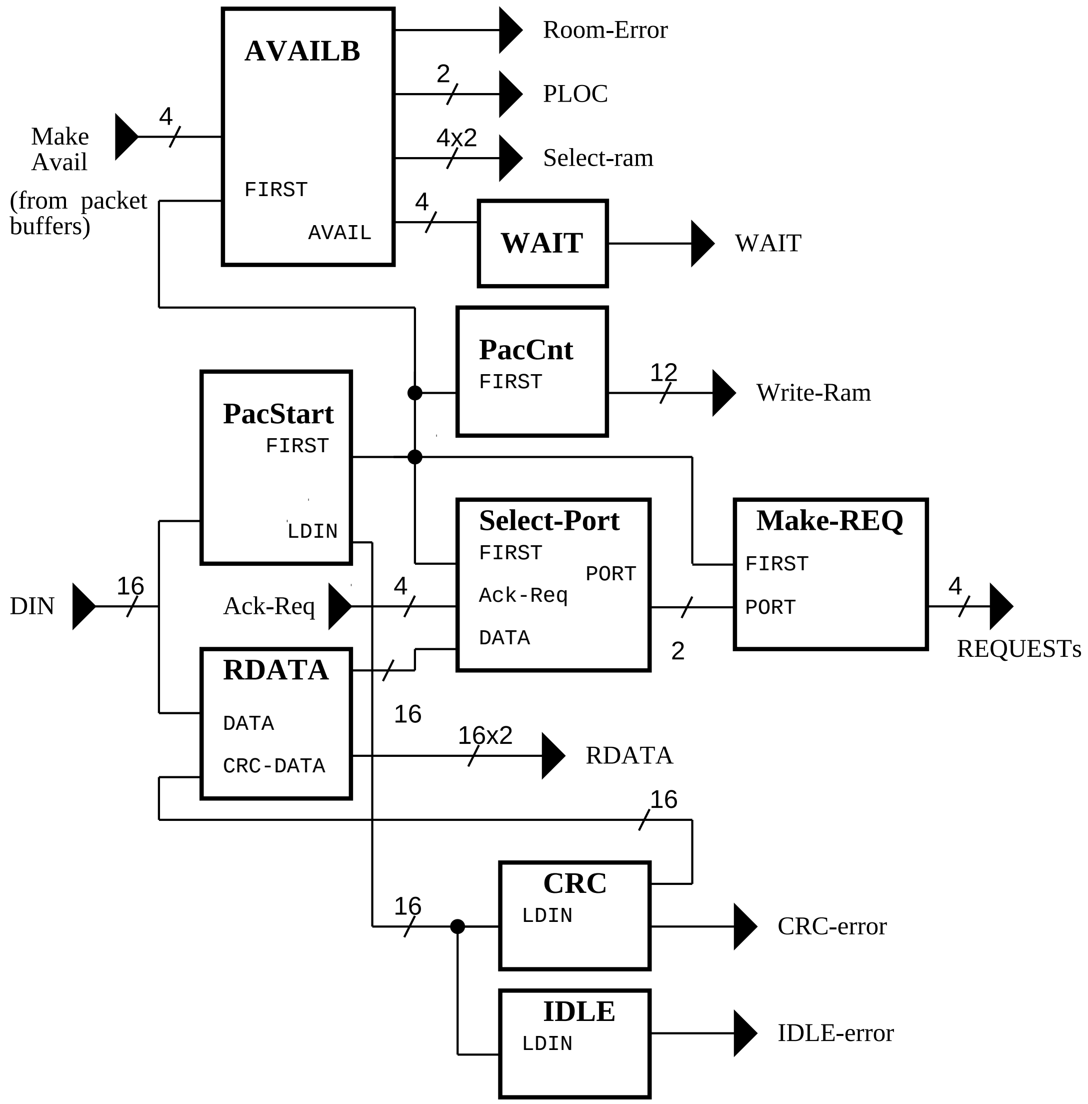


Figure 3.3: Block Diagram of the Input Port

The stage transmitting to an input port sends both data and a clock to the input port. It is this external clock (XCLK) that most of the input port runs off of. The data coming into an input port must be aligned with XCLK such that the clock rises while the data is stable. To minimize the setup and hold times of the incoming data, this data is accessed only in a limited way. When the data is brought onto the chip it goes directly into registers, and only via the outputs of these registers will the input data be sampled. To allow the register outputs to have some dependence on the previous state, scan registers will be used. These are simply registers with a two way mux on their data input. The scan registers allows the register to be loaded with either the incoming value or an internally generated value. This gives enough flexibility while adding only slightly to the data setup time.

The PACSTART module uses the incoming data to create the signal FIRST. This signal is high during the cycle when the first word of a packet is available. This signal is used by other blocks which need to know when a new packet is beginning. The signal is created by loading the incoming SOP bit into a scan register whenever a new packet could possibly begin, and loading a 0 at all other times. This module also latches the incoming data into 16 registers to produce the LDIN (Latched Data IN) bus. This bus contains the word just received and is used by the CRC and IDLE blocks to check for errors. When PaRC is generating the CRC, the generated CRC will be latched instead of the last words of the packet.

The PACCNT block counts the incoming words of the packet using a 12 bit unary counter. The output of this counter is used as the write signals for the packet buffers. The AVAILB block will produce select signals so that a packet buffer is only selected when a packet is being received.

The AVAILB block produces the AVAIL signal, which tells which of the buffers are available. When a packet starts being written into a buffer, that buffer is immediately marked as full. When the buffer signals that the packet has been sent, the buffer is marked as available. Several cycles before a new packet could be received, this block tries to choose one of the available buffers into which the next packet will be written. If none are available, then it keeps trying to choose a buffer until it is successful. If a packet starts while no buffers are available, a room error will be signaled. When a buffer is chosen, it appears in the PLOC (Packet LOCation) signal. This signal is sent to the schedulers so that when a request is made, the schedulers will know which buffer the packet is being stored in. Since we choose the next buffer shortly before the end of a packet (about 2 words before the end), PLOC will change before the packet ends. This is safe to do since we know that the scheduler will have processed the request by that time.

The reason we choose the next buffer earlier than is necessary is to allow packet buffers to be released sooner. As mentioned in the previous section, buffers can be released before their entire packet has been read out. Releasing a buffer early is beneficial because the input port may be asserting WAIT, waiting for a buffer to become available. However, we can not release a buffer too early, it must be released late enough so that a new packet stored in it will not start to be read out until the old packet is completely read out.

If AVAILB always waited until the last possible moment to choose a buffer, a buffer might be chosen just as it was released, and a packet might immediately be written into that buffer. The packet buffers would have to release their buffers late enough so that this worst-case scenario would operate correctly. By choosing the next buffer two cycles before we need it, we can guarantee that a buffer is not used in the two cycles after it is released; this allows us to release buffers earlier. When there are buffers available, a buffer will be chosen two cycles before it is needed, so obviously it will not be used for at least two cycles. However, if no buffers are available when we try to select one, a buffer which later becomes available might immediately be selected. This will still work because WAIT will have been asserted, so no packets will be received (and therefore no buffer will be used) for at least the next two or three cycles.

The AVAILB block also produces the select signals for the packet buffers. Since the write line is always active for one of the 12 words, this block must only have a buffer selected when we actually want to write that buffer.

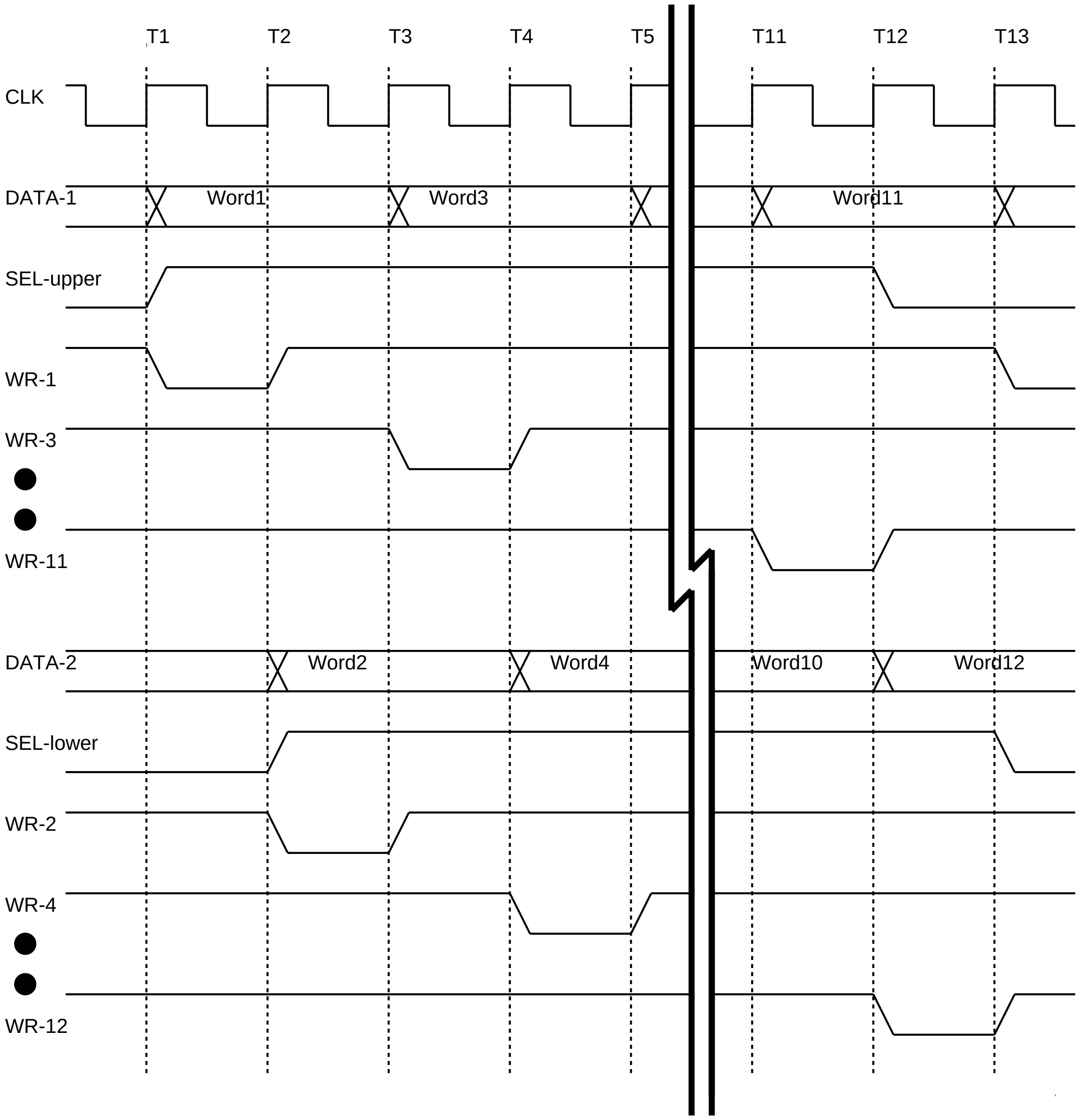


Figure 3.4: Writing into a Packet Buffer

Within a packet buffer there are two blocks of memory. The select line for each of these is controlled separately. This prevents glitches when switching from writing one buffer to writing another. As soon as the first word of a packet is received, the select line for the upper block of the buffer is asserted. The write line for the first word will be asserted for this cycle, thus writing the first word. On the next cycle the write of word one ends, and the write of the second word begins. This is when the select line for the lower block becomes asserted. This allows the second word (which is in the lower block) to be written.

The select lines stay asserted for most of the packet and writes alternate between the two blocks. The last word will be written into the lower block. When the write of this word begins, the upper select line is deasserted. This is done then because on the next cycle the write line for the first word will be asserted again, and we need to guarantee that the upper block is deselected before that write begins. On the following cycle, during which a new packet may begin, the lower block is deselected. As with the upper block, this is done a cycle before the next write to that block may occur. Figure 3.4 shows the signals used in writing to a packet buffer.

The RDATA block produces the RDATA bus, which is the data that is written into the packet buffers. Since the data going to the two blocks of memory in a packet buffer must change at different times, two buses of data are produced. One goes to the upper block of memory in every packet buffer, the other goes to the lower. When storing a packet these buses change on alternating cycles. This holds the data stable for about a cycle both before and after the end of the write pulse to a location. When not storing a packet the first RDATA bus is loaded every cycle because a new packet could begin at any time.

The value for the RDATA bus is produced by scan registers which need to be able to load data from three places. The new value for the RDATA bus usually comes from the incoming data bus. When the CRC is being generated, however, the last words of the packet are instead taken from the CRC block. Also, since the RDATA bus should only change every other cycle, when it does not need to change the registers should be loaded with their current value. This is implemented by having the incoming data go directly to the data inputs of the scan registers. As mentioned earlier, this is done to minimize the data's setup and hold times. The other input to each register is connected to a multiplexer which allows it to get either the register's current value or the CRC value.

The SELPORT block selects which port the packet should be sent to. It receives the first incoming data word from the RDATA block. The routing address is determined by selecting two bits from this word. The value of RINF (Routing-INFormation), which is a user programmable value in the control port, determines which two bits are used. The bits are selected by putting the data word into two 16-to-1 multiplexers; the upper multiplexer chooses the upper bit of the address, the lower multiplexer chooses the lower bit. Each multiplexer is separately controlled by RINF.[1] The output of the upper and lower multiplexers are used to produce the two bit PORT value, which indicates the port that this packet will be sent to.

Since the SOP bit of the first word is always 1, it is replaced by bits describing the location of this input port. So in the input to the upper multiplexer, the SOP bit is replaced with the first bit of the input port's location. Similarly in the lower multiplexer the bit is replaced by the second bit of the input port's location. This allows routing to be influenced by which port the packet arrived in.

To support the Up/Down routing mechanism described earlier, this block is able to further modify the lower bit of PORT. This block receives a signal from the scheduler for port 3 which tells it which of ports 2 and 3 has a smaller queue. It receives a similar signal from port 1 which compares ports 0 and 1. These signals are first synchronized to the local clock by being put through two registers. In order to do Up/Down routing the lower bit of PORT is put through another multiplexer. If Up/Down routing is not being done this multiplexer simply passes the bit unchanged.

[1] The value of RINF coming from the control port is used directly and is not synchronized to this ports clock domain. This implies that RINF must not be changed while packets are being received. Synchronizing RINF would have required a total of 64 registers (16 per input port). Since RINF will normally be set once during startup and then remain the same, synchronizing it was not worth the additional hardware.

If Up/Down routing is being done the bit chosen will depend on the value selected by the upper multiplexer: If the upper bit is a one, the Up/Down information coming from port 3 will be selected, otherwise the Up/Down information coming from port 1 is used. This modified lower bit is combined with the upper bit to give the final value of PORT.

The PORT signal is only valid during the first two cycles of a packet. This is because the PORT signal is produced directly from the stored first data word. After the first two cycles, the registers in RDATA which hold the first word will be overwritten with the third word. Also, since the Up/Down signals coming from the schedulers may change at any time, the registers producing the synchronized versions of these signals are constrained so that they will not change on the cycle after a packet begins. This will allow the PORT signal to remain the same for the first two cycles even if the Up/Down signals change.

When a packet begins, the MAKEREQ block will make a request to the output port indicated by the PORT signal. The block must first decode the selected port into a 4 bit unary value and store that value. It is stored using a latch which is gated by the FIRST signal. A latch is used instead of a register so that the data appears as soon as possible. Since the PORT value is held for the first two cycles, it can be safely latched in by the falling edge of FIRST which will fall at the beginning of the second cycle of the packet. When the ith make-request latches goes high, it means a request will be made to the ith port.

Since requests must be synchronized to the internal clock (ICLK), the rest of this block operates on that clock. As soon as a new packet is detected, a latch is set to 1 signifying that a request needs to be made. The output of this latch needs to be synchronized to ICLK. Normally a signal is synchronized by putting it into a register, and putting the output of that register into another register. If the first register sampled its input as the input was changing, it may take extra time for the register's output to stabilize. But with very high probability it will stabilize in time to be safely read by the second register.

Synchronizing the request signal is done slightly differently. When a packet begins a 1 is put into the first register of a synchronizer. Instead of having one register as the second stage of the synchronizer, four scan registers are used with the output of the first register being used as the select inputs of the second stage. These four registers are the request registers. A data input for the ith request register is the ith bit of the unary port value. When the 1 appears on the output of the first stage of the synchronizer, on the next clock edge the request registers will be loaded with the unary port value. So one of these registers will go high and the other three will stay low. The output of these registers are buffered and sent to the schedulers as the request signals. When one of the request signals is asserted, it stays asserted until a scheduler acknowledges the request. The second data input to each request register is used to accomplish this.

Note that the request registers are acting as the second stage of the synchronizer (synchronizing to ICLK), yet one of their data inputs (the port value) is not synchronized with ICLK. Most of the time the select signal will be low so the value of the unsynchronized data input is unimportant. (The other data input is synchronized, and is used to keep requests active until the acknowledgment arrives.) The unsynchronized data input is only sampled when a new request is about to be made. This occurs shortly after a new packet begins. More precisely, the sampling occurs at least one full ICLK cycle plus some propagation delay after the rising edge on which a new packet begins. When a new packet begins the data input giving the port value is guaranteed to get the correct value by 15 ns into the cycle. (This is why a latch is used for this value rather than a register.) This means that the data is guaranteed to be stable in time to be safely sampled.[2]

Error checking is done on packets by use of a Cyclic Redundancy Code (CRC)

[2]For the data to be safely sampled the following must be true:

$$\begin{array}{lllll} & \text{Max\_Time\_Until\_Data\_Valid} & + \text{Data\_Setup\_Time} & \leq & \text{Min\_Time\_Until\_Data\_Sampled} \\ \text{or} & 15 & + 2.5 & \leq & \text{ICLK-Period} + \text{Propagation\_Delay} \\ \text{or} & \text{ICLK\_Period} \geq 17.5 - \text{Propagation\_Delay} & & & \end{array}$$

Since the minimum ICLK period is 20ns, this will always be true.

[14]. The 32 bit CRC checksum appears in the final two words of the packet. The CRC block can either check or generate this value.

A CRC code uses polynomial division to check the integrity of messages. It treats the k-bits in a transmitted message as the coefficients of a polynomial with k terms. It divides (modulo 2) this polynomial by a known generator polynomial. If there has not been a transmission error the remainder of this division will be 0. If there has been a transmission error then, with high probability, the remainder will not be 0. The CRC can be generated by doing a similar division and appending the remainder (sometimes called the checksum) to the original message.

How effective the CRC is at detecting errors depends on the code used. PaRC uses the CRC32 polynomial, $x^{32}+x^{23}+x^{21}+x^{11}+x^{2}+1$, as its generator polynomial. With CRC codes, it is possible to prove that certain classes of errors will always be detected. When checking a packet for errors, this code will always detect an error when there are only 1, 2, or 3 erroneous bits in a packet, when there are an odd number of errors in a packet, or when all erroneous bits occur within a span of 32 bits. This last property means that if all errors in a packet occur within two consecutive PaRC data words, errors will always be correctly detected. For errors which span more than 32 bits, the probability of an error going undetected is less than 1 in 10^9. Additionally, when errors occur on only one of a port's 16 data input lines, this code will always detect the errors.

Since the CRC involves doing modulo-2 division on polynomials, it is very simple to compute using only exclusive-or (XOR) gates and registers. To compute the CRC 32 registers are needed. Initially these registers are set to 0. As the message is being received these registers store the partial result of the division. When the division is complete they will contain the remainder of the division.

The simplest CRC circuitry operates on one bit of the message at a time. It consists of a 32 bit shift register where most of the register inputs come directly from the output of the previous register. Several of the register inputs are computed by XORing the previous register's output with the output of the most significant register.

The locations with XORs correspond to terms in the generator polynomial with non-zero coefficients. This implements one step of the division wherein the generator polynomial is subtracted only if the current MSB is one. This can be expanded to allow n bits to received at a time, and the results of n potential subtractions to be computed at once. This results in each register input being the XOR of several values. PaRC receives 16 bits of data at a time, so the CRC generater was designed to consume 16 bits per cycle.

One way to check for errors is to see if, after all words are received, the remainder is 0. It is possible to check for this slightly sooner. During the last two words of the packet, PaRC compares the input data to the current upper 16 bits of the partial result. No differences will be found iff the remainder will be 0. Since we don't need to look at the remainder, we can synchronously reset the registers to 0 so that the CRC can be checked for the next packet.

To generate the CRC we could simply process the message as above, and the remainder will be the CRC. As in CRC checking, this is modified so that only the upper 16 registers need to be examined. After we have received the first 10 words of the packet the upper 16 registers will contain the first part of the remainder, and the lower 16 the second. By replacing the current CRC data input (which is meaningless anyway) with the upper part of the remainder, on the next cycle the lower part of the remainder will be shifted into the upper 16 registers.

This gives us the first (second) part of the CRC in these registers at the same time the 11th (12th) word of the incoming packet would normally be available. But we want to be able to use the first (second) part of the CRC in place of the 11th (12th) word. We would like other blocks to be able to simply latch in the CRC word instead of the incoming data word. To do this we need access to the CRC one cycle earlier. This is done by sampling the *inputs* to the upper 16 registers, instead of their outputs. This value is sent to other blocks which can latch it at the same time they would latch the 11th (or 12th) word of the packet.[3]

[3]The paths where these values are sent to the RDATA block are the longest paths in the input ports. These paths are about 4ns faster than they need to be for PaRC to run at 50 MHz.

The IDLE block checks each incoming word to see if it is identical to one of the two idle patterns (hex 5555 and 2AAA). If a word is not one of the idle patterns and it is not part of a packet, then the idle-error signal is asserted.

3.4 Scheduler

Each output port has its own scheduler which tells it which packet should be transmitted next. The major components of the scheduler are shown in Figure 3.5. The scheduler receives request signals from the input ports and processes them by storing the requests on a queue. When requests are waiting in the queue, the scheduler informs the transmitter of this by asserting MORE. While MORE is asserted, PacInfo should contain the information about the packet that should be transmitted next. After the transmitter begins transmitting a packet, it will assert NEXT. This means it wants to find out about the next packet to be transmitted.

An important goal in the implementation of the scheduler was to make it fast since time spent scheduling adds to the latency of non-blocked packets. The latency of the scheduler is one cycle: When the transmitter is waiting for a packet to send, and a request is made, on the next rising clock edge the transmitter will be able to latch in the information about that request.

The scheduler receives a request signal from each of the input ports. These signals have already been synchronized by the input ports so that they will only change at the beginning of the cycle. Since several input ports may make a request at the same time, the scheduler must choose between them. For simplicity, the scheduler uses a fixed priority scheme: The input ports are given a fixed priority (3,2,0,1) and the request with the highest priority will be selected. This scheme is slightly inequitable since whenever two ports make concurrent requests, the same port will be chosen every time. A fixed priority scheme is allowable because we can guarantee that the lowest priority port cannot be "starved." If its request is ignored in favor of one from a higher priority port, the request is guaranteed to be chosen before the higher

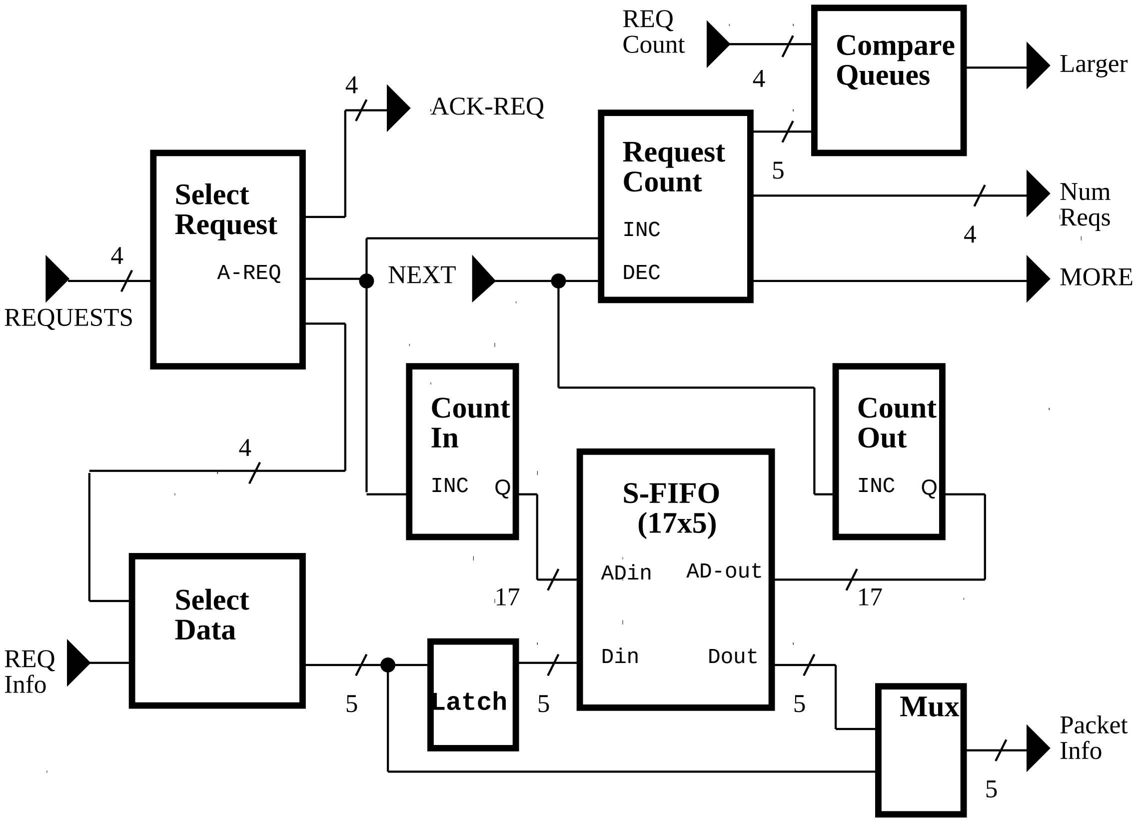


Figure 3.5: Block Diagram of the Scheduler

priority port can make another request.

A fixed priority scheme was used because of its speed. With this scheme, the requests need only go through two levels of logic (an inverter and a nand gate) to select the chosen request. A fairer scheme would have been slower and required either lengthening PaRC's clock period or pipelining the scheduler (thereby increasing the latency of non-blocked packets). The slight benefits of fairer scheduling were not worth either of these costs. For simplicity, each scheduler uses the same priority list. (*i.e.,* Every scheduler would choose port 3 over port 2 if both made requests at the same time.) Giving each scheduler a different priority list would have helped to remove some of the unfairness of a fixed priority scheme. Since the scheduling algorithm is otherwise first-come-first-served, this unfairness will not have much of an effect on PaRC's performance.

Once a request is chosen we need to select the rest of the request information from the input port that made the request. The chosen request (encoded in unary) is used as the select inputs to a mux to select the packet information bits from the correct input port. From these muxes the scheduler gets 2 bits which tell which of the input port's buffers was used to store the packet, and one bit which tells whether or not the packet is a circuit switch packet. As this is done the selection lines are encoded into two bits which indicate which input port the packet is stored in. These 5 bits, which will be stored in the s-fifo, tell all that the transmitter needs to know about the packet.

At the heart of the scheduler is the S-FIFO and its associated counters. The s-fifo is a 17 by 5 array of ram cells. One 5 bit request can be stored in each of the 17 locations. Each of these locations has its own separate write line (actually write and nwrite lines) and read line.

The scheduler maintains two counters as pointers into the s-fifo. The input counter is a 17 bit unary counter which identifies the next location to be written. The outputs of this counter are directly used as the write lines for the ram. Since this counter always points to the next location to be written, every cycle will do a write of the next available fifo location. If no request is being made during a cycle then the value written will be meaningless. When a request is made a correct entry will be written into the ram and the value of the input counter will be incremented on the next clock edge; thus preserving the entry.

Since the write of an s-fifo location will not end until after the next cycle begins, there is a 5 bit latch at the data input of the s-fifo. This holds the s-fifo data input stable during the high phase of the clock, thus keeping the data stable until well after the write has ended.

Although there can only be 16 valid entries in the s-fifo at any one time, the s-fifo has 17 locations. This is because we always do a write of the next location. By having this extra location we are able to greatly speed up, and simplify, the control of the s-fifo.

The second counter, the output counter, always points to the next request in the s-fifo. This counter value is used as the read address of the s-fifo, so that the s-fifo is always reading out the next request. This request information is passed on to the transmitter as PacInf. Since the transmitter needs this information early in its cycle, the transmitter will latch in this information at the beginning of any cycle during which it may want to start reading out a new packet. (This means that PacInf must be valid in time to be safely latched on a rising clock edge.) After the transmitter has used this information it will assert NEXT. This will cause the output counter to be incremented so that the next request will be read. (This is done several cycles before the new request is actually needed.)

Since we begin to read each s-fifo location before its value is needed, the value can be read out in plenty of time to be presented to the transmitter. This is not the case, however, when the s-fifo is empty. When a request is made to an empty s-fifo, the request would need to be written into the ram, and then read out soon enough for the transmitter to latch it in at the beginning of the next clock cycle. The s-fifo is not fast enough to do this; instead we use a cut-through technique. When the s-fifo is empty, instead of using the value being read from the s-fifo for PacInf, we use the data that will be written into the s-fifo. This allows the value of PacInf to reach the transmitter in time. Even when this is done, we still write the request into the s-fifo. On the cycle after this request is made, we will go back to using the s-fifo output.

The signal MORE is used to tell the transmitter when there are more packets to be sent. If it is high during a cycle then if PacInf was latched at the beginning of the cycle a valid request was latched.

To compute MORE the scheduler needs to know how many packets it has in its s-fifo. A third counter, this one an up-down counter, is used to keep a running total of the number of packets in the s-fifo. If this counter is greater than 0, or if a request is being made, then MORE will be asserted on the next clock edge. Instead of this third counter we could have compared the first two counters to see if they were the same. The extra counter is used because the count is also needed for up-down routing.

In up-down routing we need to determine which of ports 0 and 1 (or 2 and 3) have the shortest queue. This is done by having the scheduler for port 0 (and 2) send its current queue length to the scheduler for port 1 (3). That port can then compare the lengths to see which is greater. Schedulers 1 and 3 then send this information to the four input ports. These schedulers also send an s-fifo count to the control section so that the s-fifo count can be made visible.

3.5 Transmitter

The transmitter is the component that is responsible for transmitting packets from the chip. The scheduler gives it information about the next packet to be transmitted. It must then set up a connection to the packet buffer containing the packet, tell that buffer to begin reading out its data, and send out the data. Between packets, idle patterns must be transmitted. Figure 3.6 shows an overview of the transmitter.

3.5.1 Outgoing Data

To ensure that the transmitted data is valid for the longest possible time, the 16 bits of outgoing data come directly from 16 registers. These registers are loaded with either the data coming from the packet buffers, or with an idle pattern. This section must also produce the clock that accompanies the transmitted data. This is done by taking ICLK (which also clocks the registers) and putting it through an appropriately sized inverter. This provides a clock which rises while the data is stable. We must guarantee that the clock rises at such a time that it will provide sufficient setup and hold times.

There are a number of problems which limit our ability to maximize the setup and hold times. One is clock skew. By placing an output port's data registers and clock inverter in the same area of the chip, the clock skew between them is kept fairly small.[4] A bigger problem is the variation of delays with process, voltage and

[4]The skew of ICLK is as great as 2.3 ns between some components. But considering only the 16

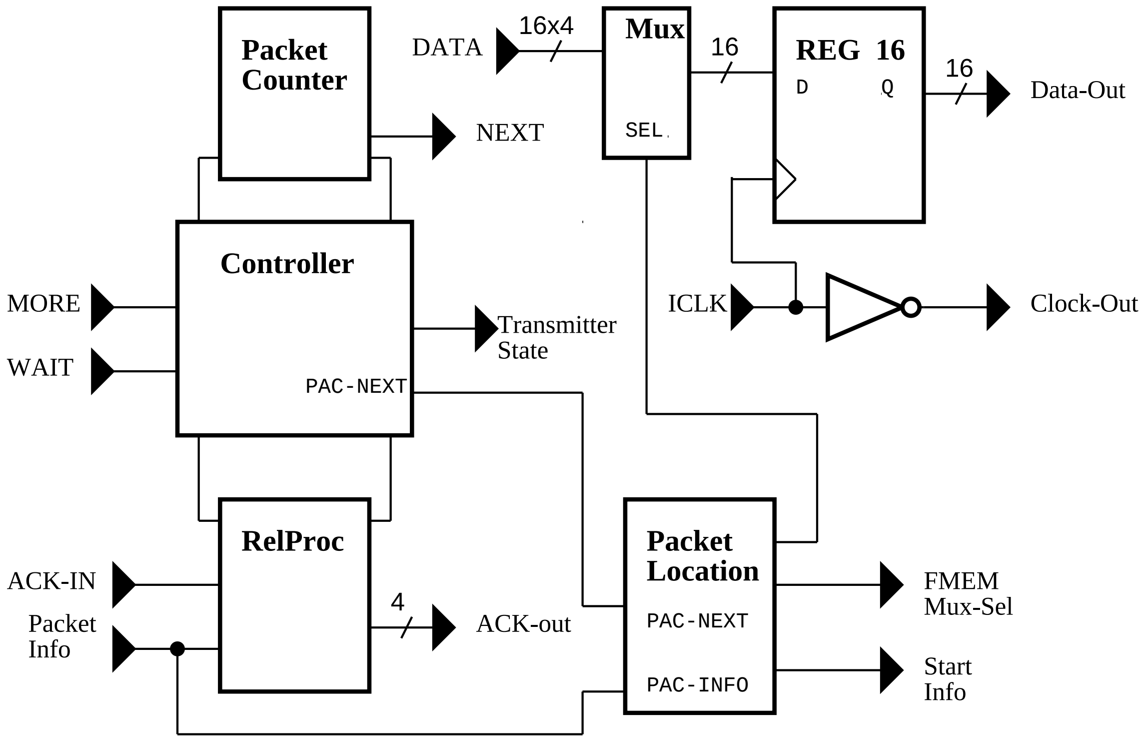


Figure 3.6: Overview of the Transmitter

temperature. The worst case delay of an element is almost four times greater than its best case delay after accounting for these three variations. The variations due to temperature and voltage are the least important of these. This is because they have less of an absolute effect on the delays than process variations, and because their changes affect the delays of all gates similarly. Process variations can account for delays varying by about a factor of three. This effect is made worse because all gates are not affected the same by process variations. Gates which pull up and those which pull down are created by different processes. So it is possible, for example, to have gates with worst case rise times on the same chip as gates with best case fall times. The worst case combinations must be considered when determining the true setup and hold times provided.

single bit registers and the clock inverter in any one output port, the skew between any 2 components is only 0.4 ns.

3.5.2 Transmitter Logic

The control section of the transmitter is made up of logic which counts how much of the current packet has been sent out, and determines when a new packet can be begun. A new packet can begin if 1) No packet is currently being sent out. 2) MORE is high, indicating that there are additional packets waiting to be sent out. 3)WAIT is low, indicating that the receiver can accept another packet. 4) This output port has not been turned off by the controller. 5) If the next packet is a circuit switched packet, the output port is not currently waiting for an acknowledgment.

To shorten the latency of packets, the transmitter is able to begin sending a packet (*i.e.,* load the output register with the first word of the packet) on the same cycle it determines that a new packet is allowed to begin. As mentioned previously, to do this the transmitter must be able to read the first word from a packet buffer in the same cycle that it finds out which packet buffer should be read from. This is done by latching in PacInf (the signals from the scheduler which tell where the packet is stored) at the beginning of any cycle during which a new packet might begin. Two bits of these latched values are sent to the select inputs of multiplexers in each of the four FMEM blocks.[5] This selects the data from one packet buffer in each FMEM block. Two more of the bits are used locally as inputs to 4-to-1 multiplexers which select between the data coming from the four FMEM blocks. In this way data from any of the 16 packet buffers can be selected.

The packet buffer in which the new packet is stored must be told to start reading out the rest of its data. To do this the address from the stored PacInf is sent to PaRC's control section, along with a signal which goes high only when a packet actually begins. The control section receives this start information from all four output ports and uses it to generate START signals for the 16 packet buffers. While a packet is being transmitted, the transmitter will assert NEXT for one cycle. This tells the scheduler to send information about the next packet to be read.

[5]These signals are the beginning of the longest paths on PaRC so it was important to minimize their delays. The Q output of the registers are buffered and sent to the two most distant FMEM blocks. The QN outputs of the registers are buffered and sent to the two closest FMEM blocks.

The RELPROC block of the transmitter can generate or process acknowledgment signals for circuit switch packets. This block receives the output port's incoming acknowledgment signal and produces four acknowledgment signals, one for each input port.[6] To be able to produce these acknowledgment signals, when a new circuit switch packet begins, this block latches in the part of PacInf which tells which input port the packet came from. When it becomes time to give the acknowledgment, this value is used to route the acknowledgment to the correct port.

If the PaRC chip has been told to generate acknowledgment signals, then a 2 cycle acknowledgment pulse will be given shortly after a circuit switched packet begins to be sent. If it is not supposed to generate the acknowledgment, then it will wait for an incoming acknowledgment. As soon as this acknowledgment arrives (*i.e.,* as soon as the incoming acknowledgment line is asserted) the outgoing acknowledgment pulse begins. The outgoing acknowledgment pulse will end 1 to 2 cycles later. Once this is done the control section of the transmitter is told that it is no longer waiting for an acknowledgment; this will allow circuit switched packets to be sent again. The start of the incoming acknowledgment pulse is used to both start and end the outgoing acknowledgment pulse. The unsynchronized version is used to start the outgoing acknowledgment pulse in order to minimize the delay of an acknowledgment passing through a PaRC chip. The synchronized version is used to end the outgoing acknowledgment pulse in order to guarantee that an acknowledgment pulse does not grow too long or too short. If the end of the incoming pulse was used to end the outgoing pulse then the pulse might get shorter (or longer) each time it passed through a PaRC chip. Using only the beginning of the pulse guarantees that the length of each pulse remains within a fixed range.

[6]When an acknowledgment signal is asserted for an input port, that input port's RELOUT output will be asserted.

3.6 Control Section

The control section performs a number of functions which affect the operation of the entire chip. In particular, it implements the control port which allows a network control system to control and monitor the performance of PaRC. The control section also keeps track of any errors that occur and keeps statistics on how the output ports are being utilized.

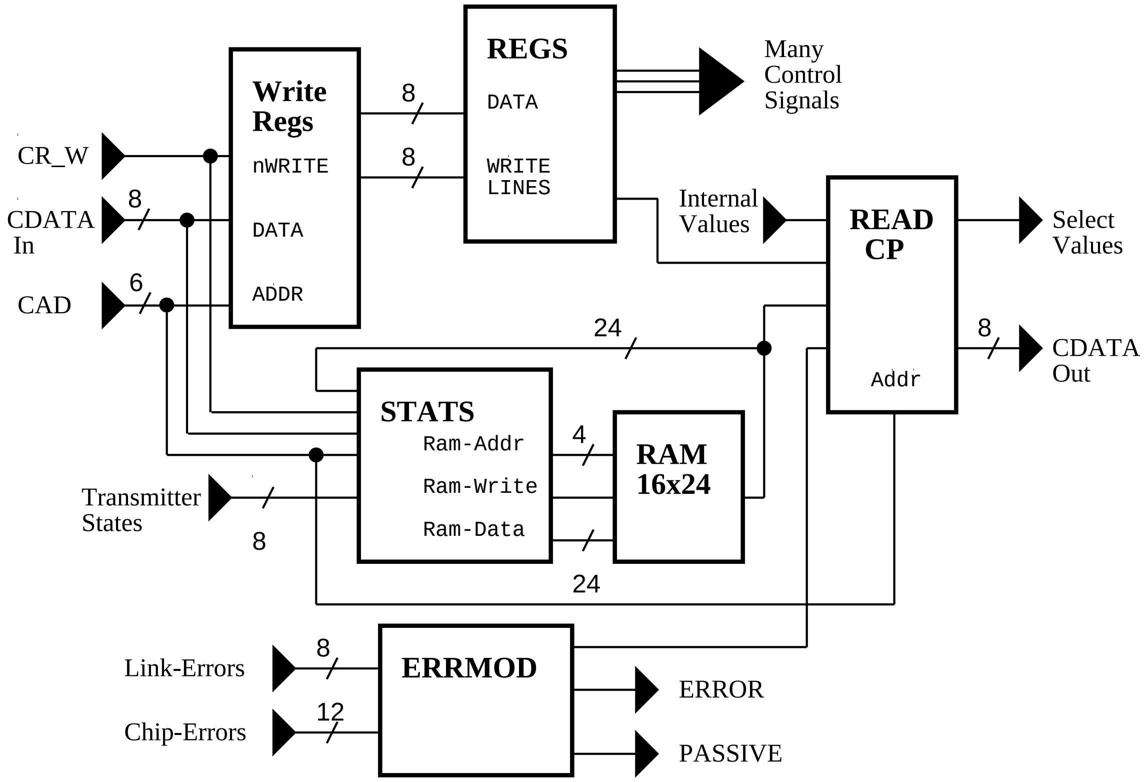


Figure 3.7: The Control Port

An overview of the control section is shown in Figure 3.7. The REGS block contains registers which store the values which control the function of PaRC. The WRITE-REGS section controls the writing of the values in REGS. The ERRMOD block keeps track of errors that have occurred, and takes any appropriate action. The STATS block keeps statistics on the use of PaRC's output ports. These statistics are stored in a 16x24 ram. The READ-CP section handles reads of the control port. The values read may reside in the control port or may come from elsewhere on the chip.

To the user, the control port looks like 64 locations which can be read or written. The first 8 of these are used for setting various values which control the operation of PaRC. The next 8 are used solely for reading out values about PaRC's current state. The final 48 are used for reading (and writing) the statistic values. See the User's Guide for details on the use of each specific location.

The control port interface is made up of 6 address lines (CAD), an 8 bit bidirectional data bus (CDATA), an output enable for the data bus (C_OE), and a write line (CR_W). A location is read when an address is given on CAD, and the output data bus is enabled. A write is done whenever the write line is brought low. The address lines should be kept stable during this time. If a specific value needs to be written, the output data bus should be disabled, and the value driven onto the bus.

The WRITE-REGS section controls the writing of the values in REGS. A write should be done whenever the write line is asserted and the value on the address bus is less than eight. The incoming write line is inverted and used as a clock to store the values needed for the write. Whenever the write line is asserted this will store the data bus, the lower three bits of the address bus, and a bit which tells if the upper three bits of the address bus were all 0. If the upper three bits are all 0 then the address is less than eight and a write to the REGS section will be done. The write is done by producing a one cycle pulse on one of the eight write lines going to the REGS section. The above data and address values are latched in so that even if the input lines change after the write begins, a valid write will still be done. This prevents a bad write from corrupting many values in the REGS block.

The REGS section contains the registers that store the control values. Each of the eight locations receives its own write line. Five of the eight locations store readable/writable values. Some locations use latches with their enable connected to the write line. Others use registers which are loaded on clock edges in which their write line is asserted. The registers take up more space than latches, but they are used because they produce outputs that only change at the beginning of a cycle. The other three locations are write only locations. When a write is done to one of these locations, some command is executed. One of these locations simply produces a pulse to clear the error counters. The other two are used to produce pulses about six cycles long. One is used as a reset command, the other is sent to the data link chips to tell them to synchronize to the PaRC chip.

The ERRMOD module keeps track of the errors that have occurred. There are 12 two bit counters used to count the number of CRC, IDLE, and ROOM errors that occur in each input port. These counters start at 0 and are incremented by one each time an input port signals a particular error. Once they reach their maximum, further increments will have no effect. The counters count using a gray code, which means that only one bit of the output will change at a time. This is done so that if one of the counters is read while it is being incremented, a valid value will still be obtained. The link errors do not use counters, but have just a single bit that says whether or not a given link error has occurred on a given output port. The error values can be reset by doing a write to the reset-error-counters command location.

This module also produces PaRC's ERROR output. This output goes high when any of the error values are non-zero. The gray encoding used by the counters prevents glitches from appearing on this signal when a counter increments. This module also produces the PASSIVE signal. When this signal is asserted the input and output ports will go passive. (*i.e.,* Output ports stop sending; input ports assert WAIT.) As mentioned earlier, PaRC can go into passive mode when an error is detected. Settings in the REGS section tell whether all errors, all errors except link gray errors, or no errors at all will cause PaRC to enter passive mode. PASSIVE is computed from the error values, synchronized to ICLK, and then sent to the input and output ports.

The READ-CP section handles reads of the control port. Mostly this involves multiplexing between the data stored in other sections of the control port. Values can be read from the REGS, ERRMOD, or STATS section of the control port. Some of the values that can be read come from the input and output ports. For these a signal is sent to the ports to select what data they send to the control port. The selected data is sent out whenever the C_OE signal is low.

There are 16 statistic values, each 24 bits long. When a read is done of a statistic location, the lower 4 bits of the address are used to determine which value is read; the upper 2 bits are used to determine which 8 bits of the 24 bit value are returned. Since the STATS module controls the ram while statistics are being collected, statistics cannot be read during this time. If they are then the value currently being output from the ram will be returned, regardless of which location was requested.

3.6.1 Statistics

The STATS section, shown in Figure 3.8, is used to keep statistics on how each output port is being used. It keeps track of how much time the outputs spend in the following states: transmitting (XMIT), idling (IDLE), waiting for a circuit switch acknowledgment (CSWAIT), and waiting for the WAIT signal to be deasserted (WAIT). These sixteen values (four per output port) are stored in a 16-by-24 ram.

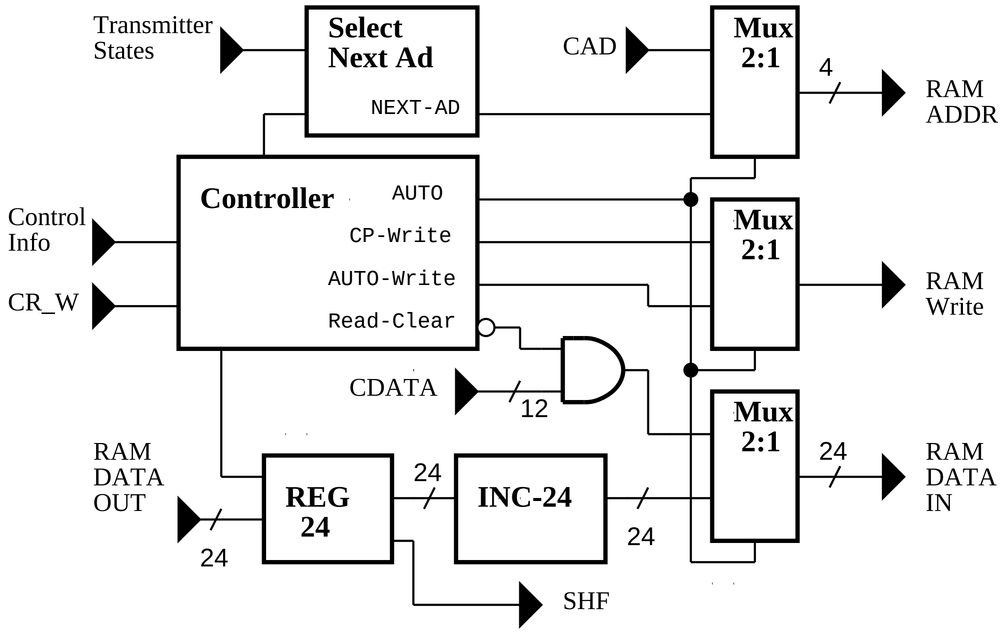


Figure 3.8: The Statistics Section

The STATS module can be in one of two modes: AUTO or Read-And-Clear mode.

When in AUTO mode the module will collect statistics by repeatedly reading and updating ram locations based upon the current state of the output ports. When in Read-And-Clear mode, the ram is under the control of the control port.

When in AUTO mode, the stats module determines the state of an output port, increments the value corresponding to that state, and then moves on to the next output port. This update process takes three cycles, so each output port is sampled every 12th cycle. (Since packets are 12 cycles long, each output port's XMIT statistic will give a count of the number of packets that port has transmitted.) During the first of the three update cycles the new ram address is latched in. The first two bits of this ram address correspond to the next port to be updated, the second two bits correspond to the state of that port. The next port to be updated is determined directly from the previous port. The state of the next port is determined in advance and stored in a register. This ensures that the next ram address is ready to be latched in during the first cycle of the update.

The new address is sent to the ram and the read of the ram begins. At the end of the second cycle, the 24 bit value returned from the ram is stored in a register. During the third cycle the stored ram value goes from the register, through a 24 bit incrementer and to the data input of the ram. Also during the third cycle the write pulse is given to the ram. At the beginning of the next cycle (which is the first cycle of a new update) the write pulse will end. The ending of the write pulse is used to trigger the load of the new ram address. This ensures that the ram address stays valid until the write has completed.

All of the signals sent to the ram first go through 2-to-1 multiplexers. When not in AUTO mode, these multiplexers allow the ram's input to be controlled through the control port. The ram's address will be taken from the lower bits of the CAD bus. The ram's write pulse is generated from the CR_W, the control port's write signal. When a write pulse appears on CR_W, it will be passed on to the ram if the address on CAD is greater than 15. (An address of 16 or more means that a statistic location is being addressed.) When not in AUTO mode the data input to the ram will normally be given the value 0. This allows ram locations to be initialized when they are written to. For testing reasons, when not in AUTO mode, the ram data can also be taken from either the CDATA bus or from the incrementer.

The 24 bit incrementer is built out of six submodules which are 4-bit incrementers. Each 4-bit incrementer takes in 4 bits of data and a carry-in signal. They each produce a "group carry" signal and 4 bits of the result. The group carry output is asserted iff all four of the data inputs to that submodule are ones. The carry-in to each submodule is asserted if all the submodules before this one have their group-carry asserted. (Additionally all carry-inputs can be forced high for testing purposes.) When carry-in is asserted a submodule will increment its data input, otherwise it will output it unchanged. The longest path through this incrementer is only 4 gates. The value going to the incrementer comes from a register which is loaded on ICLK. The time from the rising edge of ICLK until the incrementer's output is stable is less than 12.5 ns (assuming worst case process, temperature and voltage). This needs to be fast in order to meet the data setup time for the ram. The write pulse going to the ram begins based on the same rising clock edge which loads the incrementer's register. This write pulse will end one cycle later.

When collecting statistics, the statistic values must be sampled and reset regularly, or they may overflow. If the incrementer overflows it simply wraps around to 0. Since values are 24 bits long and can be incremented only on every 12th cycle, overflow could occur after $2^{24} * 12$ cycles. At 50MHz, this is approximately every 4 seconds. To make it easier to read the statistics before they overflow, PaRC can produce a Statistics-Half-Full (SHF) signal. When statistics are being collected, this signal will be asserted when a stats location is written with a value that is more than half the maximum value. When the network control system sees this signal asserted it can stop statistic collection and read out the statistic values. It can then reset the values and restart statistic collection.

3.7 Other Details

This section describes several other important characteristics of the implementation of PaRC.

3.7.1 Single Packet Latency

Minimizing the single packet latency was an important goal of this design. The latency of this design is approximately 100ns. To measure this time we will calculate the delay from when the incoming clock rises (with the first word of the packet on the input data bus) until the first word of the packet is on the output data bus and the outgoing clock rises. The time from XCLK going high until the new-packet signal is set up at the synchronizer input is under 14 ns. When this signal is detected by the first stage of the synchronizer depends on how the incoming XCLK is aligned with ICLK. This may take up to one cycle. One cycle after this the request signal will be generated, and one cycle later the MORE signal will go high, telling the output port that a packet is waiting to be sent. One cycle later the output registers will be loaded, and the first word of data will be placed on the output bus. Half way into the next cycle (when ICLK falls) the rising output clock will be generated, and will appear outside the chip within 5 ns. Summing up these times gives us $3\frac{1}{2}$ ICLK cycles, plus 19 ns, plus 0 to 1 cycles at the synchronizer input. This is less than $4\ \frac{1}{2}$ to $5\ \frac{1}{2}$ cycles, which is equivalent to 100 ± 10 ns when PaRC is running at 50MHz.

3.7.2 Critical Paths

The design goal was for PaRC to be able to run at 50 MHz. This meant that all register to register paths would have to be under 20 ns. When determining if a path meets the 20ns limit, there are two factors that must be taken into account. The first is the actual delay of the logic, the second is time lost due to clock skew. Since a single clock driver is driving the clock signal to all gates on the chip, the clock will arrive at different locations at different times. If the clock going into the register at the end of a path arrives before the clock going into the register at the beginning, then the difference in the clock arrival times (*i.e.,* clock skew) must be subtracted from the 20ns limit. Approximate clock skews are not known until after layout, but the skews are usually under 2 ns. When the chip was being designed all paths were kept under 18ns. This was to allow margin for clock skew, or for any delays that might have been longer then originally predicted.

The longest paths in PaRC are the paths used to set up the multiplexers in the FMEMs. When a new packet is begun in an output port, a packet's address is latched into registers at the beginning of a cycle. The outputs of two of these registers are sent to all four FMEMs to control the multiplexers which send data to the output port. In each FMEM these signals are buffered and used as the select input to the multiplexers. Once the select inputs are available, these multiplexers select the appropriate data and send it to the output port. The output port then uses additional multiplexers to choose between the data coming from the four ports. The chosen data then goes into a register. The length of these paths was kept under 18 ns. This was done by using some extra hardware to speed up the first part of the paths (*i.e.,* the part that produces the mux select signals). Speeding up the first part of a path was more economical since it consists of only a few signals, whereas the second part consists of a bus of 16 signals.

Although these paths go through several blocks, each path ends in the same block it began in. This means that the beginning and ending of a path will be in the same area of the chip. For this reason we expected the clock skew on these paths to be small. This turned out to be the case. After layout was done, all critical paths on the chip were under $17\frac{1}{2}$ ns, even after accounting for clock skew.

3.7.3 Clock Frequency Differences

This design does put some constraints on how closely the clock frequencies of two adjoining stages must be matched. One such constraint occurs in the reading and writing of packet buffers. A packet can be read out of a packet buffer while it is still being written in. ICLK, which is used to read out the packet, must not be too much faster than the XCLK which is used to write the packet. If it were, we could try to read part of a packet before it was written.

From the time the first word is received, it takes just under 12 XCLK cycles for the last word to be written into the packet buffer. From the time the first word is received, it will be over 2 ICLK cycles before a transmitter begins to read that packet. Reading out the last word will not begin for another 11 cycles. So to guarantee that the last word is written before it is read, the time for 13 ICLK cycles must be more than the time for 12 XCLK cycles. In other words, the XCLK period must be at most $13/12 = 1.08$ as long as the ICLK period.[7] There are a number of other interactions between ICLK and XCLK that put constraints on their relative timings, but the constraint described above is the most restrictive.

[7]This analysis uses conservative timing numbers. Doing a more precise analysis shows that XCLK can be as much as 1.13 times as long as ICLK.

Chapter 4

Test Vectors

4.1 The Need for Test Vectors

After chips are fabricated it is necessary to test them to determine if they were fabricated correctly. A set of fabrication tests is needed to check if the logic on the chip functions as the circuit design says it should. This is not the same as testing to see if the design is correct. That is a very different question. Testing to see if the design is correct would involve comparing the high level specifications of the chip to how the design actually functions. But for these fabrication tests the specifications of the design are irrelevant. This test must compare the actual silicon to the silicon that was specified by the design (regardless of whether or not that design is "correct").

This test is done by producing a series of vectors which tell what inputs should be applied to the chip, and what results the chip should produce. Automated testing equipment will cycle through these vectors in order. On each cycle it will read the test vector, apply the appropriate inputs, and check to see if the chip outputs match the predicted outputs. If the actual outputs do not match the expected outputs then the vector is said to fail. The job of producing these vectors is known as test vector generation.

There are two goals of test vector generation. The first is to make sure that vectors will fail only if the chip is defective. This goal is non trivial because there is more variance in the actual operation of the chip, than in its simulation. Consider the simple case where a data input is changed shortly before a clock input is changed. In the simulation this may work fine because the data always arrives before the clock. However, when the test is actually run there will be some uncertainty (about ± 4 ns) in when the tester actually sends any given signal to the chip. If the data is skewed late and the clock is skewed early, the data may not arrive in time and the vector will fail even though the chip is not defective. There are also large uncertainties in the delays inside the chip due to process variation. For both these cases, we must ensure that vectors will produce the same results whenever the delays fall anywhere within the allowable range. A number of methods are available to help ensure that this goal is met. For example, we can run simulations where some inputs are changed earlier or later than they should be. This imitates the skew that could be introduced by the tester, and could point out problems that tester skew could create. By using these and other tests on the test vectors, we can be fairly confident that a test vector will fail only if the chip is defective.

The second goal is to construct the vectors such that if there is a defect anywhere on the chip, then some vector will fail. This is a much harder goal to meet. This is difficult both because of the large amount of logic on the chip, and because some of that logic is hard to control directly. We must also try to use as few test vectors as possible since we can only use a limited number of them. (LSI Logic, for example, recommends limiting the number of test vectors to 16,000. If significantly more are needed, they can be added for an additional charge.) It is for these reasons that the job of test vector generation can be so time consuming.

4.2 Testing for Defects

To check for defects in the chip we need to have a model of what defects may do to the operation of the chip. A useful model is the "stuck at" model. In this model a defect is assumed to cause a node on the chip to always be zero or to always be one. The goal of test vector generation is to produce vectors that will find any single stuck-at-zero or stuck-at-one fault. Of course other types of faults are possible; for example two signals might be unintentionally connected. Also, it is possible to have more than one stuck-at fault on the same chip. Although we detect any single stuck-at fault, this does not guarantee that we will detect a problem if two faults occur. However if we can detect any single stuck-at fault, it is very probable that we will detect any type of fault or number of faults that might occur.

To detect a given stuck at fault, we must present inputs such that if that fault exists the output values will differ from what they would be if the fault did not exist. Let's first look at the simple example of testing a 2 input nand gate. We want to

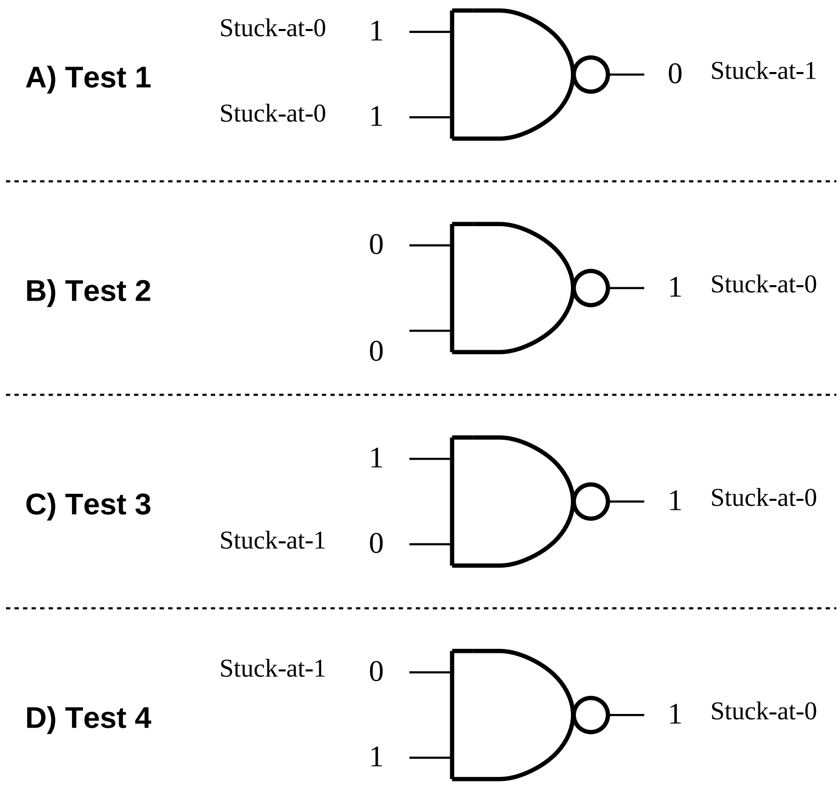


Figure 4.1: Testing a NAND gate

check both its inputs and its outputs for any stuck-at faults. For a 2 input NAND gate there are only four possible input sets. Figure 4.1 shows the four possible tests and what faults each test would detect. Part A of the figure shows the test where ones are applied to both inputs; the result should be a zero. If either input were stuck at zero, then the output would be a one, so the vector would fail. Similarly if the output were stuck at one, then the result would be incorrect and the test would fail. Thus this test checks for stuck-at-zero faults on each input, and a stuck-at-one fault on the output. Part B of the figure shows zeros applied to both inputs with the expected result being a one. This test checks only for a stuck-at-zero in the output. Even if one of the inputs was stuck at one, the output of the gate would not be affected. Part C shows the test where a one is applied to the first input, a zero to the second, and a one is expected at the output. If the second input was stuck at one, both inputs would be one, so a zero would be produced and the test would fail. If the output was stuck at zero the test would also fail. The first input is not tested; even if it had an incorrect value (zero instead of one), the output would not have changed. The final test, shown in part D, is similar to the previous test except that the inputs are reversed.

If we wanted to create a set of test vectors for this NAND gate, we would not have to include all four tests. We need only the minimal set of tests such that all faults are checked for. In this example tests A, C and D form such a set. Test B is not needed since it only detects output-stuck-at-1 faults, which two other tests also check for. Carefully choosing the tests can reduce the number needed; this is especially true with gates with many inputs. For example an n-input NAND gate can be tested using only $n+1$ of the 2^n possible inputs combinations.

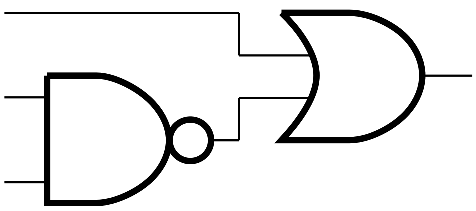


Figure 4.2: A NAND-OR circuit

In the above example we assumed that the output of the NAND gate was directly visible, but this is not always the case. Consider the case shown in Figure 4.2 where the output of the NAND gate is put into an OR gate. Now there are further constraints on our test of the NAND gate. If the top input to the OR gate is one, then the output of this circuit will always be one. Any attempted tests of the NAND gate will be useless, since the NAND gate will have no effect on the output of the circuit. If the top input to the OR gate is zero, then the output of the circuit will be the same as the output of the NAND gate. So to test the NAND gate, we must do the three tests chosen above while the upper input to the OR gate is zero. This is easy to do if the top input to the OR gate is directly controllable, it takes more effort if it is produced by other logic. When a change in a gate's output will cause a change in the output of the chip, we say that the output is observable. In general, whenever a gate is tested we must make certain that its output is observable.

When circuits also include memory elements, the task of checking for faults becomes even more difficult. When the input to a gate is influenced by the state of a register, then all tests of that gate are influenced by the previous test vectors. Before doing a test of that gate we will have to load the register with a specific value. This may take just one vector, or it may require many. If the output of a gate goes to a register, it becomes more difficult to ensure that when the gate is tested, its output is observable. If the output of the register goes directly off chip, then for the gate's output to be observable, we must simply make sure that the register is loaded during the cycle in which the test is done. A fault in the gate would then show up as an incorrect output one cycle later. Usually, however, the register does not go directly off chip, but instead feeds other logic. So to test a gate, we must also control the rest of the chip both before and after the fault is tested for. The chip must be manipulated such that if the gate has a fault, a changed gate output value will work its way through the logic and registers, and eventually cause a changed chip output. Again, this changed chip output may happen shortly after the faulty gate is tested, or it may occur many cycles later.

4.3 Test Vectors for PaRC

There are a number of problem circuits that are encountered when generating test vectors. This section will illustrate some of these by choosing some components in

PaRC and showing how they were tested.

For many of the gates in PaRC devising tests is quite easy. Consider testing the data inputs of a memory cell in a packet buffer. If we store a packet that has a one in that location, and then later read it out, that location's data lines have been tested for stuck-at-zero faults.[1] Later a packet can be stored that has a zero in that location. Reading it out will complete the test for stuck-at-one faults.

Just by receiving and sending a few packets, much of the logic in the input and output ports is tested. This is because much of the logic is used every time a packet is received and sent. Circuitry that is used less often will take more time to test. For example, testing the ram cells in the scheduler's memory will take longer than testing packet memory because a write to a scheduler's memory is done only 1/12 as often as a write to a packet memory.

Circuitry with many more inputs than outputs can be difficult to test, even if it is used often. Consider the comparators in the scheduler which are used for Up/Down routing. These comparators take as input the number of packets this scheduler has, and the number that its partner has. It produces only a single output which says which one is less than the other. Since there are so many inputs that give the same result, it is easy for a fault to have no effect on many input patterns. In fact there are gates that will have an affect on only one of the many possible input patterns. To test this component it was necessary to examine each gate in the comparator and make a list of what input values would detect the faults that could occur on that gate. This was then used to create a list of input patterns that would together detect all faults. By carefully choosing a subset of these patterns, only eleven tests were needed to cover all faults. The input to this comparator is the number of requests in each of two scheduler queues. So for each of the eleven tests, packets had to be sent to, or released from, the two output ports so that each port contained exactly the

[1] For this test a packet may be written into a buffer, held for a long time and then read out. If there is a fault on the data input line the wrong value would be written into the ram, and this wrong value would later be read out. This is a simple example of a fault not being observed until many cycles after it occurred. If the packet is never read out of memory, then the fault would remain unobserved.

number of requests needed. Since this could take many cycles, the testing of the two Up/Down comparators took approximately 500 test vectors.

In the previous examples, the circuitry could be tested without adding any additional hardware. Sometimes more control over the component is needed, so testing hardware must be added. One way to add hardware for testing is by placing multiplexers at the inputs (and/or outputs) of a component. The multiplexers will choose between the normal inputs for the component, and special inputs used only for testing. A variant of this was used for testing the statistic memory. The ram's output could already be observed through normal control port operations. Similarly it's address and write line could be controlled through the control port. Circuitry was added to allow the ram's data input to get its value from the CDATA input bus instead of from the incrementer. The CDATA bus is only 8 bits wide, while the ram is 24 bits wide. Four extra inputs were used to give a total of 12 bits of data for the ram. The 24 bit ram data input was then formed by using two copies of these 12 bits. This provided enough flexibility to completely and easily test the ram.

It is often advantageous to include testing circuitry when a component is designed. The 24 bit incrementer in the statistics section is one example. To test this without any additional hardware would have taken an inordinate number of cycles. By designing the incrementer to allow all carry-inputs to be forced high, testing was made much more efficient. Testing of the incrementer was also simplified by making use of the ability to load the ram with a value from from the CDATA bus.

Sometimes it is actually impossible to test for some faults without additional hardware. This is the case when some logic must work correctly in order to prevent a race condition. Consider a register which is occasionally disabled in order to prevent a race condition. (This occurs in the select-port section of the input port.) The enable signal could be stuck in such a way that the register was always enabled. This fault is not easy to detect because under test conditions the race may always be "won" by the appropriate signal. But this is no guarantee that the same will be true when the chip is in actual use. To test the enable input the output of the register must be made observable. This will make it possible to see when the register is loaded. One way to make the signal visible is to make it a chip output, but this could waste many outputs. In PaRC a number of hard-to-see signals were wired into a tree of EXCLUSIVE-OR gates whose result is sent off chip. Such a tree has the property that if a single tree input changes, then the output of the tree will also change. So if any one of the inputs to the tree has the wrong value, the output of the tree will be inverted, so the fault will be detected. We will have no information about which input to the tree caused the fault, but this is not a problem since we have no intention to try to debug the internals of a faulty chip.

Another place where additional hardware must be added is in the scheduler. When a request arrives in a scheduler it is written into the scheduler's s-fifo. If the s-fifo is currently empty, the request must be sent directly to the transmitter. The s-fifo write is still done, and after a short delay the written data will appear at the output of the s-fifo. The data sent to the transmitter is selected by a multiplexer which can choose either the data coming out of the s-fifo, or the data going into the s-fifo. Since the test vectors run much slower than the chip will operate, when a write is done to an empty s-fifo the two inputs to the multiplexer will become the same before the multiplexer's output is used. Yet the test vectors must somehow guarantee that the multiplexer is selecting data from the correct input. This is done by adding a chip input (nTESTMODE) that can force the scheduler input (but not the mux input) to a known state. We can use this to cause the multiplexer inputs to differ while in cut-through mode, thus allowing the multiplexer to be tested. This input is used only for testing and must be tied high during normal use.

Lastly, there was part of a component that could not be efficiently tested even if a moderate amount of additional hardware was added. This was the clear inputs in the schedulers' 17 bit unary counters. Each of these counters contains 17 registers, and only one register in a counter may be high at a time. To test the clear inputs to each register we would need to reset the counter while each register had the one high bit, and make sure that each reset took place correctly. Since these counters are only incremented once per packet, this would have taken several thousand vectors. Instead some extra circuitry was added (just two gates) so that even if a single register's clear input is stuck, the counter will be correctly reset anyway.

The process of test vector generation for PaRC was time consuming since first the entire chip had to be examined to determine what needed to be done to test each gate. Then test vectors had to be created that tested all the gates. These test vectors had to conform to certain limitations so that the test equipment will be able to perform the tests correctly. The test vectors themselves also had to be tested to ensure that they would not fail on a correctly fabricated chip. All total, some 13,000 test vectors were generated for PaRC.

Chapter 5

Conclusions

5.1 Future Work

The design of the PaRC chips described in this document has been completed, and working chips have been received. In the future it may be desirable to build another version of PaRC to take advantage of improvements in technology. There are several modifications to PaRC that any future versions should consider.

One change would be to implement PaRC in a BiCMOS technology instead of CMOS. A BiCMOS chip has a CMOS core of gates, with bipolar gates on the periphery. The bipolar gates are faster than CMOS, and would be used mostly for input and output functions.[1] By using bipolar for receiving and sending data we could output data at a higher frequency than PaRC currently can. This might allow us to move the functionality of the Data Link Chip onto the PaRC chip. For each PaRC chip used this would save the board space used by four DLCs, as well as the TTL/ECL level converters that are currently needed between PaRC and the DLCs. Since data is being sent out at a higher frequency we would be able to provide a much higher bandwidth per port. We would also have the option of keeping the same bandwidth per port but either reducing the number of pins or increasing the number of ports.

A future version of PaRC might include a retry mechanism. A retry mechanism

[1] Bipolar technology is not used for the entire array because it has significant power and density disadvantages.

means that when an error is detected in a packet, the receiver would ask the sender to send it again. So instead of merely detecting errors, PaRC would try to correct them. The disadvantage of providing retry is that packets must be kept around until the sender is sure that the receiver has received it correctly. This takes away buffer space that could be used for storing incoming packets. Given a limited amount of space for buffering, this would reduce the throughput of the chip. We did consider including retry in PaRC. However, the links being designed were expected to be as reliable as a backplane; so the gains from having retry did not seem worth the costs. Any future versions of PaRC would probably be done in a smaller geometry process which would allow more gates per die. This would allow more packet buffering to be added, so including retry would not have much of an effect on throughput.

A final possible change would be incorporating additional routing paradigms. The current paradigms work very well when the network can be divided into a small number of stages such that a message will only pass through a given stage once. In these networks messages tend to pass through only $O(\log n)$ switches. Butterfly networks and fat trees are two examples of this. Networks where messages make many more hops, such as in a mesh, do not fit this structure well. PaRC can only be used to implement very small networks of this type. There are a number of ways the routing mechanism could be extended to support this. One is to allow each PaRC to be given an ID equal in length to the routing data. Each port would look at certain bits of a packet's routing data, compare it to the same bits in the ID, and choose the output port based on how they compare. This would give enough flexibility to support other networks, such as a multidimensional mesh.

5.2 Summary

Even without these changes, PaRC is a very powerful building block for constructing networks. We designed PaRC to be able to be used in a large variety of high performance networks. There are a number of features of PaRC that make this possible.

PaRC can be used to implement networks with both high throughput and low latency. Each port of PaRC provides 800 Mbits/second of bandwidth, and much of that bandwidth is usable. Also, when a packet comes into PaRC and its output port is available, the latency of that packet will be at most 5 $\frac{1}{2}$ cycles (110 ns).

A number of aspects of the design contribute to achieving this performance. In particular, the design of the packet buffers and scheduler help tremendously. The packet buffers do not constrain packets which arrived via the same port to be sent out in the same order in which they arrived. In addition, the packet buffers allow many buffers from the same port to be read simultaneously. These properties greatly reduce the amount of blocking that occurs since an output port will never be blocked because another output port is using resources that it needs. In order to effectively use the large number of independent packet buffers, the scheduler uses a fifo of requests. This provides a very fast scheduler; the scheduler can process a request on the same cycle the request is made. The buffers and schedulers allow a packet to be read out before the entire packet has been received. This virtual cut through is essential in minimizing the latency of packets. Using scheduler fifos also provides a very equitable scheduling strategy, coming very close to a first-in-first-out ordering.

The interface also helps to make effective use of the bandwidth. The interface does not need a dedicated line to indicate when packets begin. Given a fixed interconnection bandwidth, this increases the bandwidth available for data. Also the interface protocol does not require idle time between packets; so if packets are available, they will be sent out continuously. Of course engineering PaRC to run at a high speed, 50 MHz, helps by giving each port a high raw bandwidth.

PaRC provides a great deal of flexibility for the interconnection networks built out of it. Much of this flexibility comes from its routing mechanisms. Many types of networks can be built out of PaRC. Networks with up to 2^{15} distinctly addressed nodes are supported. The basic mechanism allows routing to be based on any two bits of a packet's routing information. Routing can also be influenced by which port the packet used to enter the chip. For networks which have several paths to a destination, PaRC allows routing decisions to be based upon congestion. This can help to even out the flow of messages. The butterfly and the fat tree are just two examples of the types of networks which can be built out of PaRC.

Interconnection flexibility is also provided in ways other than routing. PaRC uses an asynchronous clocking scheme, wherein it assumes that each input port operates and receives data asynchronously from the rest of the chip. This allows PaRC to be used in a machine regardless of whether a synchronous clock is available. This, along with PaRC's flow control mechanism, allows the use of a wide variety of link lengths. Lastly, PaRC provides a statistics facility so that we are able to determine how well a network is performing.

PaRC also provides a high degree of reliability. PaRC includes 32 bits of CRC error checking in every packet. This is used to ensure that messages have not been corrupted. PaRC also checks for errors when packets are not being received. This helps to prevent packets from being unknowingly lost if a start of packet indicator is missed. Checking for errors when packets are not being received also helps by detecting problems as soon as they occur. This may allow us to stop the use of a bad connection before any data has been lost.

PaRC is also able to provide a fast acknowledgment that a packet has been received. By using circuit switched packets when an acknowledgment is needed, we can provide an acknowledgment more quickly than by other methods. (When an acknowledgment is being sent back to the sender, it takes under 15ns to pass through a PaRC chip.) This method also reduces traffic since no extra packets need to be sent to provide the acknowledgment.

The PaRC chips described in this report have been fabricated by LSI Logic, and have successfully run the test vectors. Several Monsoon processor boards have been built, each incorporating a single PaRC chip which operates at full speed for receiving and sending tokens. PaRC has been tested in these boards and so far no problems have been found. Early in 1991 4x4 switch boards built using a PaRC chip will be available. These will allow larger versions of Monsoon to be built.

Appendix A

PaRC User's Guide

A.1 Introduction

PaRC is a 4 by 4 packet routing switch which has been designed and fabricated as a CMOS gate array. PaRC receives packets via one of its four input ports, stores the packet in an on-chip buffer, and eventually transmits the packet via one of its four output ports. Each input port operates asynchronously and has enough buffering to store four packets. PaRC can operate at speeds up to 50 MHz, giving each port a raw bandwidth of up to 800 Mbits per second. The buffering and scheduling algorithms implemented in PaRC allow PaRC to make effective use of this bandwidth, while providing a low latency (as low as $5\frac{1}{2}$ cycles). The routing mechanism in PaRC allows it to be used in a wide variety of networks. In addition, PaRC provides a mechanism whereby a processor can quickly receive an acknowledgment when a packet it sent has been received. This user's guide will first give an overview of the chip, and then describe the details of the function of the chip. It will not describe the details of the implementation, except where such details are necessary to understand the behavior of the chip. Also, this document will only give a sketch of why PaRC functions as it does. For more information on this, and on the implementation, see the body of this document.

These are the components which make up PaRC, along with their primary function:

- **4 input ports**: receive packets
- **16 packet buffers** (4 per input port): store 1 packet per buffer
- **4 output ports**: each made up of:

  - **transmitter**: transmit packets
  - **scheduler**: determine which packet should be sent next

- **1 control port**: used to control the operation of PaRC

The input port is the section of PaRC that receives incoming packets. Each input port has 4 packet buffers associated with it. When the input port detects the beginning of an incoming packet, it chooses an empty buffer and writes the packet into that buffer. As it does this it checks the packet for errors by the use of a CRC code. It must also determine which output port the packet should go to, and inform that output port's scheduler that a packet is waiting and where it is being stored. The input port must also generate a flow control signal, so that it does not receive more packets than it has room for.

Each packet buffer can store exactly one packet. For performance reasons a buffer must have several characteristics. It must be able to begin reading out a packet while that packet is still being written into the buffer. This helps decrease the latency of packets. Also, it is important that each buffer can be read by any output port at any time, regardless of which other buffers are currently being read. This will greatly decrease blocking in PaRC and thus improve both throughput and latency.

Each output port consists of a scheduler and a transmitter. The scheduler keeps track of all the packets which wish to use the output port and chooses which packet will be transmitted next. If two or more of the waiting packets were received via the same input port, the scheduler guarantees that they are transmitted in the same order in which they arrived. When the transmitter is ready to send a packet, it first finds out from the scheduler where the next packet is being stored. The transmitter then reads the packet out of a packet buffer and transmits it to the next stage of the network. If the packet is one which requires an acknowledgment (such packets are called circuit switched packets), the transmitter must also keep track of the packet's input port and begin to look for an acknowledgment that the packet has reached its destination. When this acknowledgment arrives, the transmitter must pass back an acknowledgment through the packet's input port.

The control section has two main functions. One function is to keep statistics on the performance of the network. The other is to act as an interface to the network control system. This interface can be used to control and monitor the performance of PaRC. In particular, the control section contains a series of registers each bit of which controls different aspects of PaRC. These bits will be referred to throughout this guide, however the details of the reading and writing of these registers will be described only in the section on the control port.

A.2 Input Port

A.2.1 Input Data Bus

An input port is responsible for receiving packets into PaRC. It receives packets over a 16 bit data bus called, for input port x, DINx.[15..0]. It also receives a clock, called XCLKx, which is used to clock in the data on DINx. The rising edge of this clock must occur while the data is stable. The input port will be operate based on this incoming clock. Each port's XCLK is assumed to be asynchronous from the other XCLKs and from ICLK, PaRC's main clock. However, each XCLK must have a frequency close to ICLK's (within 10%).

A PaRC packet is 192 bits long, made up of 12 16-bit words. The first 16 bit word is a header. The header contains all the information needed for routing the packet. The next 9 words are data and the final 2 words normally contain the CRC value which is used for error checking. Packets cannot be interrupted or canceled. Therefore once the first word of a packet is received, the next 11 words will be received on the following cycles.

WORD	USE
0	Header
1	Data word 0
2	Data word 1
3	Data word 2
4	Data word 3
5	Data word 4
6	Data word 5
7	Data word 6
8	Data word 7
9	Data word 8
10	CRC word 0
11	CRC word 1

PaRC Packet Format

During each cycle the input port will receive a word on its DINx bus, and it must decide whether or not this word is part of a packet. This is done by using bit 15 as a start-of-packet (SOP) bit. If the input port is not currently receiving a packet, and if the new word has 0 in its start-of-packet bit, then this word is not part of a packet; it is an idle word.

When a word arrives where this bit is 1, that word is taken to be the first word of a packet. Since packets cannot be interrupted, the next 11 words must be the rest of the packet, and can use bit 15 as a data bit. Once the packet ends, the input port begins to look at the start-of-packet bit again. There is no need for an idle word between packets; a new packet may begin immediately after the previous one has completed.

When an input port is not receiving a packet, it should be receiving one of two idle patterns: x5555 and x2AAA. (Note that bit 15 of each of these words is a 0.) If the input port receives a word that is not the start of a new packet (*i.e.,* bit 15 is a 0.), but is not an idle pattern either, then an idle error (**IDLERR**x) occurs. The control port is informed that this error has occurred, and the input port continues operating, treating the word as an idle word. Checking for idle patterns can be disabled by setting the CHK-IDLE bit [bit 4 of the STATUS1 register] in the control port to 0.[1]

Idle patterns are used to allow the input port to catch almost all transmission errors. Here are the possible transmission errors, and how they will be detected:

- Error occurs inside of packet (*i.e.,* any bit except SOP bit). This will cause a CRC error.
- Error turns SOP bit from 1 to 0. This will cause an idle error.
- Error occurs in bits 0-14 of idle pattern. This will give an idle error.
- Error occurs in bit 15 of idle pattern (*i.e.,* turns SOP bit from 0 to 1.) This will create a packet which will cause a CRC error.

This makes it very unlikely that an undetected error will occur. And when errors do occur, it will be easy to pinpoint their location since we can tell where the error first appeared.

In addition to the start-of-packet bit, the header also has two other uses. Bits 13..0 are used as a routing address to determine which port the packet will be sent

[1] Only one idlerr is asserted for each burst of bad idle words. This may be more meaningful than counting each individual error, especially since the idle-error counter only counts up to three.

to. This provides 2^{14} distinct routing addresses. The section on routing describes how these bits are used. Bit 14 is used as a "circuit switch" indicator. If this bit is 1 then then the packet is treated as a circuit switched packet (CSP). Optionally, the circuit switch feature can be turned off by setting the ALLOW-CSW bit [bit 0 of the STATUS2 register] in the control port to 0; when this is done no packets will be treated as circuit switched packets. Bit 14 can also be used as an additional bit of the routing address. Usually this is only done when ALLOW-CSW has been set to 0.

Header Format		
Bit 15	Bit 14	Bits [13..0]
1	CSP/ROUTE-DATA.14	ROUTE-DATA.[13..0]

Also, it is possible to "turn off" an input port. This should be done, for example, before a link talking to an input port is replaced or resynchronized. Each input port has a nSTOP bit in the control section. When this bit is 1, packets are received normally. When it is set to 0, the packet currently being received (if any) will be accepted normally, but no more packets will be accepted until the bit is returned to 1. The nSTOP bits are found in the nSTOP register of the control section. Bit x of nSTOP.[3..0] is the stop bit for input port x.

Each input port only has enough buffers to store four packets. The flow control mechanism should prevent packets from arriving when there are no buffers for them (see Section A.2.5). If a packet arrives when there is no room for it the control port is notified that a room error (**ROOMERR**x) occurred. The input port will read in the rest of the packet normally and then simply discard it.

A.2.2 CRC

Error checking is done on packets by use of a Cyclic Redundancy Code (CRC). PaRC uses a 32 bit checksum and accumulates values using the CRC32 polynomial.[2] As they are received, the header and the 9 data words are accumulated to give a checksum. This checksum is the 32 bits that should appear in the final two words of the packet: CRC0 and CRC1. CRC0[15..0] are bits [31..16] of the checksum. CRC1[15..0] are bits [15..0] of the checksum.

[2]The polynomial used is $x^{32} + x^{23} + x^{21} + x^{11} + x^2 + 1$.

With CRC codes, it is possible to prove that certain classes of errors will always be detected. When checking a packet for errors, the CRC32 code used will always detect an error when there are only 1, 2, or 3 erroneous bits in a packet, when there are an odd number of errors in a packet, or when all erroneous bits occur within a span of 32 bits. This last property means that if all errors in a packet occur within two consecutive PaRC data words, errors will always be correctly detected. For errors which span more than 32 bits, the probability of an error going undetected is less than 1 in 10^9. Additionally, when errors occur on only one of a port's 16 data input lines, this code will always detect the errors.

When a packet arrives, the input port will check to see if the CRC is correct. If input port x detects an error then a CRC error (**CRCERROR**x) occurs. When this happens the input port notifies the control section, but continues operating normally. CRC checking can be shut off by setting the CHECK-CRC bit [bit 1 of the STATUS1 register] in the control port to 0. PaRC can also be told to generate the CRC; this is done by setting the GEN-CRC bit [bit 0 of the STATUS2 register] in the control port to 1. When this is done the input port will ignore the last two words of incoming packets and replace them with the correct CRC words. Note again that the two words which would otherwise have been the CRC are ignored; an input port will not begin to look for a new packet until after these two dummy words have been received. Also, the value of the CRC control bits are used by a packet only as it arrives. If, for example, GEN-CRC is 0 while the packet is being received, but later changes to 1, a CRC will not be generated for the packet.

PaRC may also be configured to use only one 16 bit CRC word. This is done by setting the CRC2 bit [bit 2 of the STATUS1 register] to 0. If this is done the first 11 words of the packet are used for generating the checksum. The next word is expected to be bits [31..15] of the generated checksum. PaRC can check or generate CRCs in this mode, based on the GEN-CRC and CHECK-CRC bits described above. This is available to provide some error checking in the event that more data needs to be put into each PaRC packet. Use of the CRC1 and CHECK-CRC bits allows PaRC to be configured to have packets with 9, 10 or 11 words of data (with 2, 1, or 0 words of CRC, respectively).[3]

Normally only zero or one of GEN-CRC and CHECK-CRC need be set. If both are set PaRC will generate a CRC and check this generated CRC. An error (CRCERR) would be found only if there were a serious problem (*e.g.,* setup/hold time violation

[3]Additionally, bits in the header which are not used for routing could be used as data bits.

or a defective chip).

A.2.3 Routing

When a packet arrives, the input port must decide which of the 4 output ports should transmit the packet. This is where the 14 (or 15) ROUTE-DATA bits from the header are used. Each PaRC chip can be made to look at different bits of ROUTE-DATA. This allows networks of size $2^{14} = 16\text{K}$ to be supported (32K if no circuit switched packets are needed).

The control section of PaRC contains an 8 bit register called RINF. This controls which bits of the header are used in determining the destination of a packet. The destination is a two bit number which designates one of the 4 output ports (numbered 0 to 3). Each bit of this destination is determined separately. The 4 higher order bits of RINF control the selection of the higher order bit of the destination. The 4 lower bits of RINF control how the lower bit of the destination is selected.

Normally each of the 4 bit values within RINF designates one location in the header, and the bits from the designated locations are concatenated to form the destination. If RINF[7:4]=x and RINF[3:0]=y then the bit 1 of the destination is ROUTE-DATA.x and bit 0 is ROUTE-DATA.y. For example: if RINF = x10, then the lower two bits of the header are used as the routing address.

There are 2 RINF values that are treated differently: E and F. A RINF value of F does not select bit 15 of the header (it is always 1 anyway), but instead selects a bit from the input port's location. F in the upper part of RINF means use the upper bit of the input port's location, while F in the lower bit means use the lower bit of that port's location. This allows routing to be influenced by which port a packet arrived in. For example, in port 2 (= b10) an F in the upper nibble of RINF means the upper bit of the destination is always 1. An F in the lower nibble means the lower bit of the destination is always 0.

The value E is only treated specially when it appears in the lower part of RINF. E means that "UP/DOWN" routing is being done. In this mode, the upper bit of the destination is chosen first. This narrows down the possible output ports to two. The destination is then set to the port (among the two possibilities) with the shortest request queue. (Actually, since there is some delay, it is sent to the port which had the shortest queue 2 or 3 cycles ago).

When the value E appears in the upper part of RINF, it is treated normally; *i.e.,* it causes ROUTE-DATA.14 to be used as the upper bit of the destination. When this is done Circuit Switching would usually be disabled, as bit 14 is also used to indicate a circuit switched packet.

Also, the value of RINF should not be changed while packets are being received, as this may cause unpredictable results. To change this value while the network is in use, you must first guarantee that no packets are being sent to the input ports. This may be done by asserting all of the WAIT signals, or by temporarily stopping the output ports that are sending packets to this port. [See below for information on how to do these.]

The following table summarize this information, and the next table gives some examples.

ROUTING INFO

8 bits (X,Y): 4 bits each for upper and lower bits

X == get upper routing bit from Xth bit of header.

Y == get lower routing bit from Yth bit of header.

Exceptions:

X == xF: set upper bit to be the upper bit of that port's location.

Y == xF: set lower bit to be the lower bit of that port's location.

Y == xE: Choose between ports 0 and 1 (if upper bit is 0; else 2 and 3)
based on which output port has fewer packets waiting to use it.

RINF Examples	
RINF	**Function** (where X,Y are not one of the special values)
10	Routes based on lowest two bits.
XY	Routes packets based on bits X,Y.
XX	Routes packets to port 3 or 0 based on bit X. [= 4x2 routing]
FF	Sends packets out same port they came in.
FY	Acts as two parallel 2x2 routers: [= 2*(2x2) routing] Packets coming in on upper ports(3,2) go out one of the upper ports. The value of bit Y determines which one. (Likewise for lower ports).
XE	Does up/down routing Based on bit X choose between upper or lower ports. Then choose the output port with the shortest queue.
FE	Acts as two parallel 2x2 "equalizers:" Packets from upper ports sent to the upper port with shortest queue. Packets from lower ports sent to the lower port with shortest queue.

A.2.4 Buffer Use

There are four packet buffers associated with each input port. When PaRC is reset each of these buffers is marked as empty. When the input port begins to write a packet into one of these buffers, it is marked as full. When that packet is mostly read out, it is marked as empty, and a new packet can then be stored in that location.

The state of these buffers may be determined by reading locations C and D in the control port. These locations are called BIU20 and BIU31. BIU20 gives the Buffer-In-Use information for the buffers in ports 2 and 0. Each bit corresponds to one of the buffers and is high iff that buffer is in use.

BIU20							
bit 7	6	5	4	3	2	1	bit 0
BIU2.3	BIU2.2	BIU2.1	BIU2.0	BIU0.3	BIU0.2	BIU0.1	BIU0.0

BIU31							
bit 7	6	5	4	3	2	1	bit 0
BIU3.3	BIU3.2	BIU3.1	BIU3.0	BIU1.3	BIU1.2	BIU1.1	BIU1.0

where,
BIU$X.Y$ = BUFFER-IN-USE bit for buffer Y of port X, and is high iff that buffer is in use.

A.2.5 Flow Control

Flow control is needed on a connection (also called "link") to prevent packets from arriving when there are no buffers available to hold them. In PaRC, input port x produces the signal WAITx which must be sent "backwards" from each input port to the transmitter which sends data to that input port. When an input port brings its WAIT signal high, the transmitter which is sending packets to it should not send any new packets until the cycle after WAIT goes back low.

There are 2 modes for determining WAIT: Normal mode, and LWAIT mode. The normal way that WAIT is produced is that it is high only when all four of an input port's buffers are in use. This is the ideal signal since it will make optimal use of an input port's buffers. However this mode will work correctly only if the connection between the transmitter and the input port is not too long. If the transit time between transmitter and input port is too long, the WAIT signal will not be received in time to stop the next packet from being sent.

In order for WAIT to go high in time to stop the next packet from being sent, the following must be true.:

$T_{stop-next-packet} > T_{data-transit} + T_{produce-wait} + T_{wait-transit}$
where,

$T_{stop-next-packet}$ = Time from when word 0 is sent (*i.e.,* when DOUT.16, the clock generated by the output port, rises) to when the received WAIT must go high in order to guarantee that the next packet will not be sent. (In PaRC this is 9 cycles.)

$T_{data-transit}$ = Time from when word 0 is sent to when it is received at the input port.

$T_{produce-wait}$ = Time from when word 0 is received by the input port to WAIT goes high. (15 ns)

$T_{wait-transit}$ = Time from when WAIT is sent by the input port to when it arrives at the output port.[4]

From this we calculate that when 2 PaRCs are communicating, normal mode can only be used if the transit time from the transmitter's DOUT to the input port's DIN plus the transit time from the input port's WAIT to the receiver's INFI pin is less than 8 cycles. This mode should be used wherever possible since it will provide more efficient utilization of buffers than will LWAIT.

For longer connections, we need WAIT to go high sooner. This is done by entering LWAIT mode by setting the USE-LW bit (bit 2 of the STATUS2 register) in the control section. WAITx will now be high whenever there are only 0 or 1 buffers available in input port x. So when an input port raises WAIT, it can still safely receive one more packet. This allows us to add over 11 cycles to the $T_{stop-next-packet}$ time given above. This will allow links much longer than we expect to use.

The disadvantage of using LWAIT is that it allows available buffers to remain unused. This can happen at two times:

- When the number of available buffers drops from 2 to 1. WAIT may go high in time to stop the next packet, thus leaving one of the buffers unused.[5]
- When the number of available buffers rises from 0 to 1. There is now an available buffer, but it will not be used until a second buffer becomes available.

The generation of WAIT in LWAIT mode can be modified to attempt to avoid these situations by setting FANCY-LW (bit 3 of the STATUS1 register) to 1 (while keeping USE-LW at 1). When this is done the generation of WAIT is affected in the following ways:

[4] $T_{data-transit}$ may not be the same as $T_{wait-transit}$. If a DLC (Data Link Chip) is used between two PaRCs then $T_{data-transit}$ will be longer since the data must be latched in by the DLC chip and serialized, while the WAIT signal can be passed on untouched.

[5] This will tend to happen on links which are just long enough to require the use of LWAIT.

- When the number of available buffers drops from 2 to 1, the rising edge of WAIT will be delayed for 2 cycles.
- When the number of available buffers rises from 0 to 1, WAIT will go low for approximately 2 cycles.

Both of these will allow the transmitter to send a packet (if it has one) into the fourth, previously unused, buffer. When WAIT first goes high, the first change should delay the rise of WAIT long enough to allow the transmitter time to send a packet into the fourth buffer.[6] With the second change, when all buffers are full and a packet leaves, WAIT will go low for a short time, thereby allowing a new packet to be sent to take its place. For both of these modifications, there is only a small window of time when a packet will be able to be sent into the only empty buffer. If no packet needs to be sent during that time, then the buffer will remain empty. So the fourth buffer will be most effectively used during times of heavier traffic, and that is exactly when it is needed most.

The FANCY-LW bit has no effect when LWAIT is 0. It is expected that when using LWAIT, FANCY-LW will always be used. Only in the very rare cases when there are extremely long links would it need to be turned off.

Occasionally it is useful to force the input ports to assert WAIT. For example, if we wanted to turn off the input ports of a PaRC we should first assert all WAITs so that the transmitters talking to those input ports will stop sending packets. When the WAIT-ALL bit in the control port [bit 3 of the STATUS2 register] is set to 1, all WAITs will be forced high. Also, all WAITs may be forced high if the chip enters "Passive Mode" after encountering an error. This will be described in the documentation on the control section.

A.3 Output Port

Output ports are made up of 2 components: a scheduler and a transmitter. The scheduler is responsible for determining which packet should be sent next. The transmitter is responsible for transmitting that packet.

[6]This is done at the expense of removing two cycles from the allowable round-trip link transit time.

A.3.1 Scheduler

The job of the scheduler is to tell the transmitter which packet is to be transmitted next. It does this by maintaining a FIFO queue of packets which want to use its output port. When the transmitter wants to send another packet, the scheduler will tell it which buffer the packet is stored in and whether or not that packet is a circuit switched packet. Since the queue is FIFO, packets going to the same port will be sent off the chip in the same order in which they arrived, even if they arrived via different input ports. However, packets which arrived on different ports within 2-3 cycles of each other may be transmitted in any order.

Each scheduler keeps a running count of how many packets are in its queue. The lower four bits of these counters may be examined by reading locations E and F in the control port. These locations are called SI20 and SI31. SI20 gives this information for the schedulers in ports 2 and 0; SI31 for ports 3 and 1.

SI20							
bit							bit
7	6	5	4	3	2	1	0
SI2.3	SI2.2	SI2.1	SI2.0	SI0.3	SI0.2	SI0.1	SI0.0

SI31							
bit							bit
7	6	5	4	3	2	1	0
SI3.3	SI3.2	SI3.1	SI3.0	SI1.3	SI1.2	SI1.1	SI1.0

where,
SIx.**[3..0]** are the lower 4 bits of the scheduler counter for output port x.

This count is also used in UP/DOWN routing. On each cycle the count for output port 3 is compared to that of output port 2. If output port 2 has fewer waiting packets, then packets that want to go to an upper port will be sent to output port 2; else they will be sent to output port 3. A similar comparison is done for output ports 1 and 0.

A.3.2 Transmitter

The transmitter of output port x sends out its data over a 16 bit bus which is called DOUTx.[15..0]. It also generates an output clock called DOUTx.16. This clock is generated from the falling edge of ICLK and rises when the data on the DOUT bus is stable.

The format of packets on this bus is identical to the packet description given in the discussion of the input port. Whenever the transmitter is not sending a packet, it will generate an idle pattern. Normally the idle pattern will alternate between x5555 and x2AAA. But if the TOG-IDLE bit [bit 5 of the STATUS1 register] is set to 0, only the idle pattern x5555 will be used.

Output port x receives the flow control signal INFIx.1 (also called WAITINx). When this signal goes high the transmitter will not begin to send any new packets. (Any packets already being sent will continue normally.) Since PaRC must synchronize this signal, a packet may still start during the 2 to 3 cycles after it is brought high. Similarly, no packets will be begun during the 2 to 3 cycles after this signal goes low.

Circuit Switched Packets

If the sender of a packet wants to be informed when the packet has arrived, the sender can designate the packet a circuit switched packet. Circuit switched packets are sent through the network in the same manner as normal packets. The difference is that as a circuit switched packet passes through a switch, the transmitter keeps a "back pointer" to where the packet came from. These back pointers can then be used to send an acknowledgment to the packet's sender. When a transmitter sends a circuit switched packet, it enters circuit switch mode (CSmode) and remembers which input port the packet arrived through. It then begins to watch its INFI0.x input (also called RELIN.x) for an acknowledgment signal, which is indicated by an active high pulse. This input is driven by the RELOUT signal of the input port this transmitter is sending to. When this acknowledgment arrives, the transmitter exits CSmode and sends out an acknowledgment signal on the RELOUT line of the remembered port. (This acknowledgment is also a single high pulse.) While waiting for an acknowledgment, a transmitter can continue to send out normal packets. But if a second circuit switched packet needs to be sent out during this time, the transmitter will generate idles until the first circuit switched packet's acknowledgment is received. The next circuit switched packet can then be sent.

The transmitter can be set up to generate an acknowledgment, instead of waiting for one. This is done by setting the GEN-ACK bit [bit 1 of the STATUS2 register] to 1. When this is done, the transmitter will generate the appropriate acknowledgment as soon as it starts to transmit a circuit switched packet.[7] Note that the ALLOW-CSW bit is not used in the output port. If a packet is determined to be a circuit switched packet when it arrives, it will be treated as such, even if ALLOW-CSW is later set to 0.

Other Details

On each cycle each transmitter characterizes the use of its output link during that cycle. A link is considered to be in one of four states:

Idle: No packets being sent, or waiting to be sent.

Xmit: A packet is being sent.

CSWait: No packet is being sent because we are waiting for an acknowledgment, and a circuit switched packet is to be sent next.

Wait: No packet is being sent, but one would be sent if WAIT were low.

This information is called XINF and is used by the control port for keeping statistics. These values can be obtained by reading location 7 of the control port. This location gives the following values:

XINF			
bit 7..6	5..4	3..2	bit 1..0
XINF3.[1..0]	XINF2.[1..0]	XINF1.[1..0]	XINF0.[1..0]

where,
XINFx**.[1..0]** = the XINF value for output port x.

[7] It is expected that this bit will be set for the next-to-last stage of the network, and the final stage will have ALLOW-CSW set to 0. This will cause the acknowledgment to be generated when the packet leaves the next-to-last stage of the network. The acknowledgment can be generated at that time because once a packet is sent to the last stage of the network, no new packets will be able to get in front of it.

Encoding of XINF	
XINF	Meaning
00	WAIT
01	CSWAIT
10	IDLE
11	XMIT

Also, it is possible to turn off an output port. When this is done, the transmitter will complete any packet it was in the process of sending, but will not start to send any more packets. Packets can still be routed to this port, but the packets will not be transmitted until the port is turned back on. To turn off an output port, its bit in the nSTOP register [location 1 in the control section] must be set to 0. The nSTOP bit for output port x is bit $(x + 4)$ of the nSTOP register.

A.3.3 ICLK

ICLK is the clock which PaRC uses as its main clock. ICLK may be running at any frequency up to 50MHz. ICLK is assumed to be asynchronous to the incoming XCLKs. But the frequencies of each XCLK must be close to that of ICLK. Each XCLK is assumed to have a period that differs from ICLK's period by no more than 10%. ICLK is used both as the main internal clock and also to generate the DOUTx.16 clock that is output by each transmitter.

A.4 Control Section

The control section performs a number of functions which affect the operation of the entire chip. In particular, it implements the control port which allows a network control system to control and monitor the performance of PaRC. The control section also keeps statistics on how the output ports are being utilized. To the user, the control port looks like 64 locations which can be read or written.

A.4.1 Reset

PaRC can be brought into a known state by resetting it. This can be done either by a pin reset or by a reset command. A pin reset is done by bringing the nRESET line low. A reset command is done by writing any value to location 5 of the control port. When PaRC is reset all packets currently stored, being received, or being transmitted will be lost. Once the reset ends, each input port will be ready to receive new packets, and will have 4 available buffers to store them in. During reset, each output port will begin to send an idle pattern. After the reset ends, the output port will be in the idle state and will have no packets waiting to use the port. The only values which may be retained during a reset are in the control port. If statistics were being gathered while the reset occurred, then the statistics values are no longer valid; otherwise these values will be retained during a reset. When a command reset is done, the values in the writable control registers (locations 0-4), and the error counters will be retained. During a pin reset these values will be reset to their default state.

The nRESET input is asynchronous; when this line is brought low the reset will begin to take effect almost immediately. It should be held low for at least 4 cycles. The reset can be considered over, and PaRC full ready for use, two cycles after nRESET goes back high. When a reset command is given the reset will begin several cycles later, and will last for 6 cycles. The reset can be considered complete 12 cycles after the command was given.

A.4.2 Control Locations

The last 48 locations are used solely for statistics and will be described later. The first 16 locations are used for a variety of purposes. Most of these locations have already been mentioned. The following table summarizes these locations:

PaRC Registers 0 - F			
Location (hex)	Type	Default Value (hex)	Register Name
0	RW	10	RINF
1	RW	FF	nSTOP
2	RW	00	STAT INFO
3	RW	3E	STATUS1
4	RW	01	STATUS2
5	C	-	PaRC Reset
6	C	-	Error Reset
7	R/C	-	XINF/Sync-Link cmd
8	R	00	CRC Error
9	R	00	IDLE Error
A	R	00	ROOM Error
B	R	00	LINK Error
C	R	-	BIU20
D	R	-	BIU31
E	R	-	SI20
F	R	-	SI31

Where Type means:

RW = Can read and write this value.

R = Can only read this value.

C = "Command Location" Doing a write to this location causes a command to be executed. The value written is ignored.

The default value is the value the register is loaded with when a pin reset is done.

Errors

Previous sections have described how CRC, Idle, and Room errors are detected by an input port. When one of these errors occurs, the control port increments the appropriate error counter. For each error type there is a two bit error count for each of the four ports. When one of the CRC, Idle, or Room error locations are read, the four error counts for that error will be output. These three error locations can be interpreted as follows:

ERROR Counters							
bit							bit
7	6	5	4	3	2	1	0
P3EC1	P3EC0	P2EC1	P2EC0	P1EC1	P1EC0	P0EC1	P0EC0

where:

PnEC1	PnEC0	errors from port n
0	0	no errors
0	1	1 error
1	1	2 errors
1	0	3 or more errors

The only other errors that can occur are link errors. Each output port may be directly connected to another input port. However, when PaRCs on different boards communicate, data-link-chips (DLCs) will often be used. A link transmitter will receive an output port's data, multiplex it onto fewer wires, and transmit the data between boards. A link receiver will receive the data, demultiplex it, and present it to an input port. Each link transmitter can generate two types of errors called link-error and link-gray. (Produced by the link chip outputs TSYNCERR and TGRAYERR.) The PaRC chip receives information about these errors on the inputs LINKERR.[3..0] and LINKGRAY.[3..0]. The ith bit of each of these buses should come from the link chip connected to the ith transmitter. A rising edge on one of these inputs is used to indicate an error.

For each of the link and gray errors, the control port only has a one bit counter. (Once a link chip produces an error, it will not signal another error of that type until it has been reset.) This bit is 0 if no such error has occurred, and 1 if the error has occurred. These bits can be examined by reading the LINK-ERROR location. The value produced can be interpreted as follows:

Link Error Information			
bit 7	6	5	bit 4
LinkGray.3	LinkGray.2	LinkGray.1	LinkGray.0

Link Error Information			
bit 3	2	1	bit 0
LinkErr.3	LinkErr.2	LinkErr.1	LinkErr.0

where,

LinkGray.x = 1 if there has been a gray error from link chip x.

0 if there has not been a gray error from link chip x.

LinkErr.x = 1 if there has been a link error from link chip x.

0 if there has not been a link error from link chip x.

A PaRC chip can be made to ignore gray errors. This is done by using the CHK-GRAY bit (bit 4 of the STATUS2 register). When this bit is set to 0, any gray errors will be ignored.

PaRC produces an ERROR output. This bit will go high whenever an error has been detected, and it will stay high until the error counters are reset. When ERROR goes high, the four error locations should be read to determine what type of error occurred. Note that if an error occurs while we are not checking for that type of error (*e.g.,* a Gray error while CHK-GRAY is 0), then ERROR will not go high.

When the control port is informed of an error it may put the chip into "Passive Mode." When in passive mode, output ports will not begin to transmit any more packets, and all input ports will assert WAIT.[8] Whether or not passive mode is entered depends upon the value of the PASS-ERR and PASS-GRAY bits (bits 6 and 5 of the STATUS2 register). If a Gray error has occurred and both PASS-ERR and PASS-GRAY are 1, then the chip will be put into passive mode. If any other type of error has

[8]Passive mode is entered in an attempt to minimize the data lost. For some errors it may prevent any data from being lost at all.

occurred and PASS-ERR is high then the chip will go into passive mode.[9] The chip will remain in passive mode until either the error counters are reset, or PASS-ERR and PASS-GRAY are changed such that passive mode is no longer indicated.

Command Locations

There are three locations which are used for issuing commands to PaRC. These locations are called command locations. When a write is done to one of these locations, PaRC will perform a certain command. For command locations, the command is executed whenever the location is written to; the value written is ignored. The first of these locations is the PaRC reset command (location 5), and has already been described. The second is the error reset command (location 6). When this command is given all the error counters will be reset. The final one is the sync-link command at location 7. Writing this location will cause the synclink output to go high for 6 cycles. This output will be used to resynchronize the DLC transmitters that are receiving data from PaRC. Note that location 7 is also used to read the XINF value. When a read is done the XINF value will be returned, when a write is done the synclink command is given. Writing this location will have no effect on the XINF values.

Status Registers

The bits of the Status registers (locations 3, 4) have been described throughout this document. The following tables summarize the use of these registers.

[9] Gray errors indicate a non-fatal errors in the link chip, which means that the data being transmitted is still correct. By treating Gray errors specially, we have the flexibility to continue operating after a Gray error, but go passive on all other errors.

Status1 register			
Bit	Default	Register	meaning if bit is high
0	0	Gen-CRC	Generate CRC for incoming packets.
1	1	Use-CRC	Check CRCs in incoming packets.
2	1	CRC2	Use 2 word CRCs.
3	1	Fancy-LW	Use Fancy LWAIT modifications to LWAIT.
4	1	Chk-Idle	Check incoming idle patterns.
5	1	Tog-Idle	Toggle output idle patterns.

Status2 register			
Bit	Default	Register	meaning if bit is high
0	1	Allow-CSW	Allow Circuit Switched packets.
1	0	Gen-Ack	Generate an ack when a CSP is transmitted.
2	0	Use-LW	Generate WAIT using the LWAIT definition.
3	0	Wait-All	Assert all WAIT outputs.
4	0	Chk-Gray	Check for Gray errors from the links.
5	0	Pass-Gray	Allow passive mode to be entered on Gray errors.
6	0	Pass-Err	Enter passive mode when an error is detected.

A.4.3 Statistics

The statistics section is used to obtain information on how the output ports (and hence their associated links) are being utilized. For each output port we keep track of four statistics, based on the current state of the output port. The value of the XINF signal is used to characterize the state of the output port. As described earlier, these four states are: transmitting (XMIT), idling (IDLE), waiting for a circuit switch acknowledgment (CSWAIT), and waiting for the WAIT signal to be deasserted (WAIT). The statistic values will give us an indication of how much time the output ports spend in each state.

The statistics section compiles these statistics, and stores them in a 16x24 ram. They can be read out through the control port. The STAT-INFO register is used to control the statistics generation. STAT-INFO is a 4 bit register. Two of these bits are only used for testing and must be set to a specified value[10]:

STAT INFO			
bit			bit
3	2	1	0
Mask-SHF	0	0	Stat-Mode

The statistics section can be in one of two modes. When the Stat-Mode bit is 0, it is in Read-and-Clear mode. During this mode the statistic values can be read or they can be reset to 0. Since statistics are 24 bit values, and the control port bus is only 8 bits wide, it takes 3 reads to receive an entire statistics value. When reading a statistics value the upper two bits of the address determine whether the lower, middle or upper byte of the statistic is read. The next two bits determine which port the statistic will apply to. The last two bits determine which of that port's statistics is read.

Each location can be identified as SCpsb. The p identifies the port number (0-3), s identifies the state (X,I,C,W), and b identifies the byte (L,M,U). The Lower byte is bits [7..0] of the value, the Middle is bits [15..8], and the upper is [23..16]. The following table shows how each byte may be accessed.

[10]Bit 2 is used for testing the incrementer. If it is high, the incrementer will increment each nibble of its input. If Bit 1 is high, the statistics section will stay in Read-and-Clear mode. However, instead of clearing locations, writes to the ram will store either the value coming from the incrementer (if bit 0 is high), or a value obtained from the control port's data bus (if bit 0 is low).

Location	Value
10	SC0XL
11	SC0IL
12	SC0CL
13	SC0WL
14	SC1XL
15	SC1IL
16	SC1CL
17	SC1WL
18	SC2XL
19	SC2IL
1A	SC2CL
1B	SC2WL
1C	SC3XL
1D	SC3IL
1E	SC3CL
1F	SC3WL

Location	Value
20	SC0XM
21	SC0IM
22	SC0CM
23	SC0WM
24	SC1XM
25	SC1IM
26	SC1CM
27	SC1WM
28	SC2XM
29	SC2IM
2A	SC2CM
2B	SC2WM
2C	SC3XM
2D	SC3IM
2E	SC3CM
2F	SC3WM

Location	Value
30	SC0XU
31	SC0IU
32	SC0CU
33	SC0WU
34	SC1XU
35	SC1IU
36	SC1CU
37	SC1WU
38	SC2XU
39	SC2IU
3A	SC2CU
3B	SC2WU
3C	SC3XU
3D	SC3IU
3E	SC3CU
3F	SC3WU

Statistic Locations

When the statistics section is in Read-and-Clear mode, doing a write to any statistic location will reset all 24 bits of that statistic to 0. (The value written is ignored.)

When the Stat-Mode bit is set to 1, the statistics section enters AUTO mode, and begins to compile statistics. The statistics section determines the state of an output port by looking at XINF. It then reads that state's statistic value from statistic memory, increments it, and writes the new value back into memory. This process takes three cycles, after which it is repeated for the next output port. Since there are 4 output ports, each one is sampled every 12 cycles. Since the packet length is also 12 cycles, each port's XMIT statistic gives a count of the number of packets that were transmitted by that port.

In normal use the statistics will be put into Read-and-Clear mode and 16 writes will be done to reset all locations. Then the statistics section will be put into AUTO mode to collect statistics. It will later be returned to Read-and-Clear mode so that all statistics can be read. Values can then be reset to 0, and statistic collection can be begin again.

When collecting statistics, the statistic values must be sampled and reset regularly, or they may overflow. Since values are 24 bits long and can be incremented only on every 12th cycle, overflow could occur after $2^{24} * 12$ cycles. At 50MHz, this is approximately every 4 seconds. To make it easier to read the statistics before they overflow, PaRC can produce a Statistics-Half-Full (SHF) signal. This signals goes high when a statistic value that is being incremented is more than 2^{23} (*i.e.,* more than half of its maximum value). It will drop back low when any one of the statistics values are reset. This signal is only produced when the Mask-SHF bit (bit 3 of the STAT-INFO register) is 0. If Mask-SHF is set to 1, SHF will always be 0.

A.4.4 Control Port Interface

The control port consists of 64 locations which can be read and/or written via a simple protocol. It is implemented using several signals:

CAD.[5..0]: The address being operated upon.

CR_W: The write line (active low)

CDATA.[7..0]: Bidirectional data bus

C_OE: Output enable (active low) for CDATA bus

CDATA is the bidirectional bus on which data is sent to and from the control port. When C_OE goes high the CDATA bus will be tristated within time $\tau_{cdata-z}$. A read is done whenever C_OE is low, and CAD has a stable value. If CAD is changed after C_OE is brought low, then after time τ_{read} CDATA will have the correct value. The value of τ_{read} depends on the location being read and is listed in the chart below. If C_OE is brought low as, or after, CAD is stabilized then the time until CDATA has the correct value is the maximum of τ_{read} after CAD is stable, and $\tau_{cdata-en}$ after C_OE goes low.

Remember that the 48 statistic locations can only be read while in Read-and-Clear mode. An attempt to read or clear a statistic location while statistics are being collected will be ignored.[11] Also, remember that the values from the error locations (8,9,A,B) and the info locations (7,C,D,E,F) simply reflect the current internal state of PaRC. As such they may change (asynchronously) while they are being read.

Writes are done by asserting the address on CAD, the data on CDATA, and then pulsing CR_W low. Write pulses must last at least time τ_{wp-min}, and be separated by time $\tau_{wp-recover}$. The address must be set up $\tau_{addr-setup}$ before the write pulse begins and held until $\tau_{addr-hold}$ after it ends. The data must be set up $\tau_{data-setup}$ before, and held $\tau_{data-hold}$ after the beginning of the write pulse. When a write is done, it will take 2 or 3 cycles from the beginning of the write pulse for the write to complete.

When a write is issued, it will occur regardless of the value of C_OE. So if a write is being done to a command location, it is not necessary to bring C_OE high and drive the CDATA bus.

[11] Reading a statistic location while statistics are being collected will actually return a portion of that statistic location which the statistics section is currently incrementing.

Control Port Timing

CDATA Timing *1

$\tau_{cdata-z}$	15 ns	Time from C_OE rises until CDATA bus is high-Z.
$\tau_{cdata-en}$	15 ns	Time from C_OE falls low until CDATA bus is stable.

Read Timing *1

$\tau_{read-cntrl}$	30 ns	Read time for Control registers (locations 0,1,2,3,4).
$\tau_{read-error}$	30 ns *2	Read time for Error registers (locations 8,9,A,B).
$\tau_{read-stats}$	30 ns *2	Read time for Info locations (locations 7,C,D,E,F).
$\tau_{read-info}$	40 ns	Read time for STATS locations.

Write Timing

$\tau_{addr-setup}$	10 ns	Time addr should be stable before CR_W goes low.
$\tau_{addr-hold}$	10 ns	Time addr should be stable after CR_W goes high.
$\tau_{data-setup}$	10 ns	Time data should be stable before CR_W goes low.
$\tau_{data-hold}$	10 ns	Time data should be stable after CR_W goes low.
$\tau_{wr-recover}$	4cycles	Minimum time between start of write pulses.
τ_{wp-min}	1cyc.+10ns	Minimum pulse width on write line.

NOTES:

*1 = These times assume an output loading of 35pf or less on the CDATA bus. If the loading is greater the delay will be longer.

*2 = These values merely reflect the current internal state of PaRC; as such they may change (asynchronously) while they are being read.

Bibliography

[1] Arvind, K. Ekanadham, and D. E. Culler. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. In *CONPAR 88*, Manchester, England, 1988.

[2] G. A. Boughton. Data Link Chip. Internal Memo, Computation Structures Group, Massachusetts Institute of Technology, Cambridge MA, March 1989.

[3] W. J. Dally. Virtual Channel Flow Control. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[4] G. L. Frazier and Y. Tamir. The Design and Implementation of a Multi-Queue Buffer for VLSI Communication Switches. In *Proceedings of the IEEE Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, October 1989.

[5] G. Grafe and J. E. Hoch. The Epsilon-2 Multiprocessor System. To appear in *Journal of Parallel and Distributed Computing*, December, 1990.

[6] C. F. Joerg. Design of a Packet Switched Routing Chip for the Dataflow Supercomputer. Bachelor's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge MA, May 1987.

[7] C.E. Leiserson. Fat Trees: Universal Networks for Hardware-Efficient Supercomputing. In *IEEE Transactions on Computers*, vol. C-34, No. 10. October 1985.

[8] R. S. Nikhil. Id Nouveau Reference Manual, Part I: Syntax. Technical Report, Computation Structures Group, Massachusetts Institute of Technology, Cambridge MA, April 1987.

[9] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge MA, August 1988.

[10] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[11] M. C. Pease, The Indirect Binary n-Cube Microprocessor Array. *IEEE Trans. on Computers* vol. C-32, No. 12, Dec. 1983

[12] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, vol. 28, No. 1. January 1985.

[13] K. M. Steele. Implementation of an I-Structure Memory Controller. Technical Report LCS/TR-471, MIT, January 1990.

[14] A. S. Tanenbaum. Computer Networks. Prentice-Hall Inc., Englewood NJ, 1988

[15] Thinking Machines Corporation. Connection Machine Model CM-2 Technical Summary. Thinking Machines Technical Report HA87-4, Cambridge, MA, April 1987.

[16] S. G. Younis. The Clock Distribution System of the Multiprocessor Emulation Facility. Technical Report LCS/TR-366, MIT, June 1986.